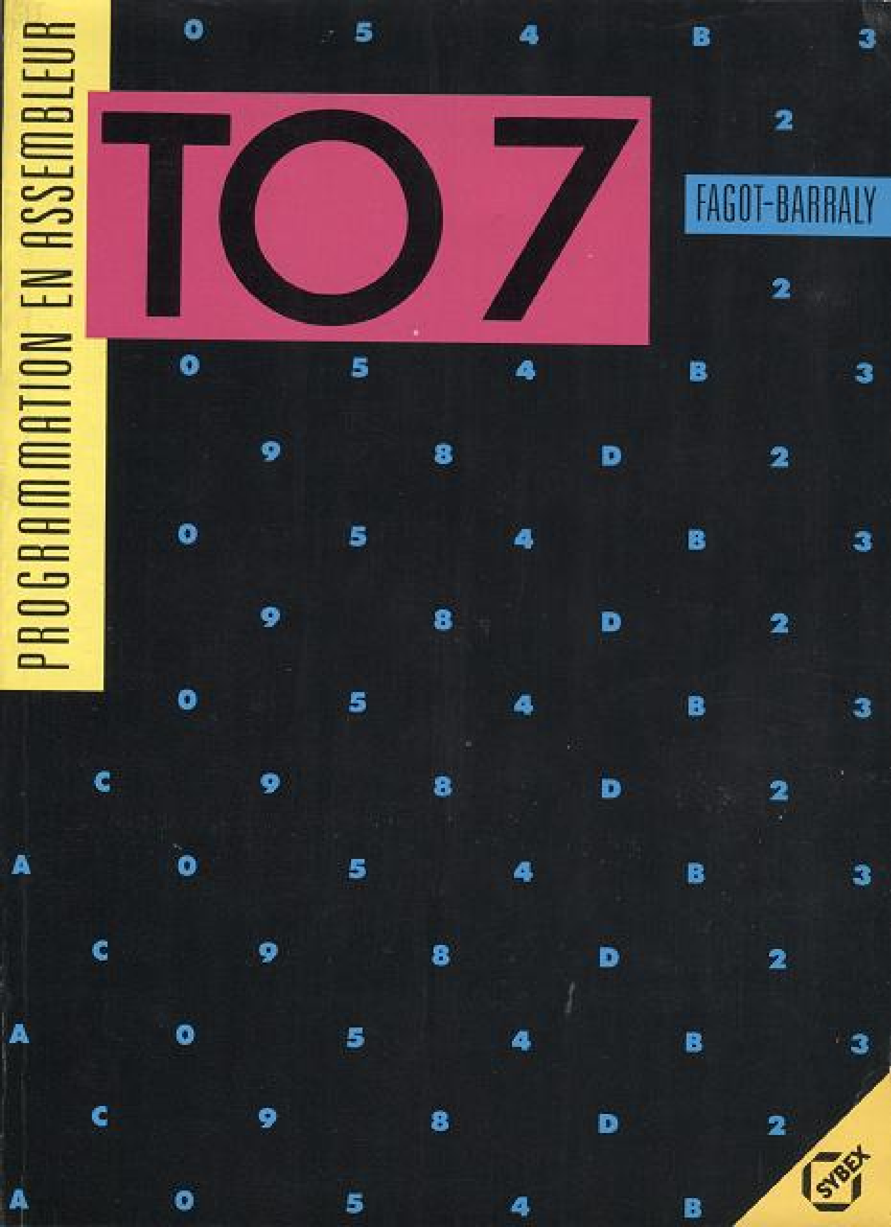


PROGRAMMATION EN ASSEMBLEUR
TO7
FAGOT-BARRALY
SYBEX

TO7

PROGRAMMATION EN ASSEMBLEUR

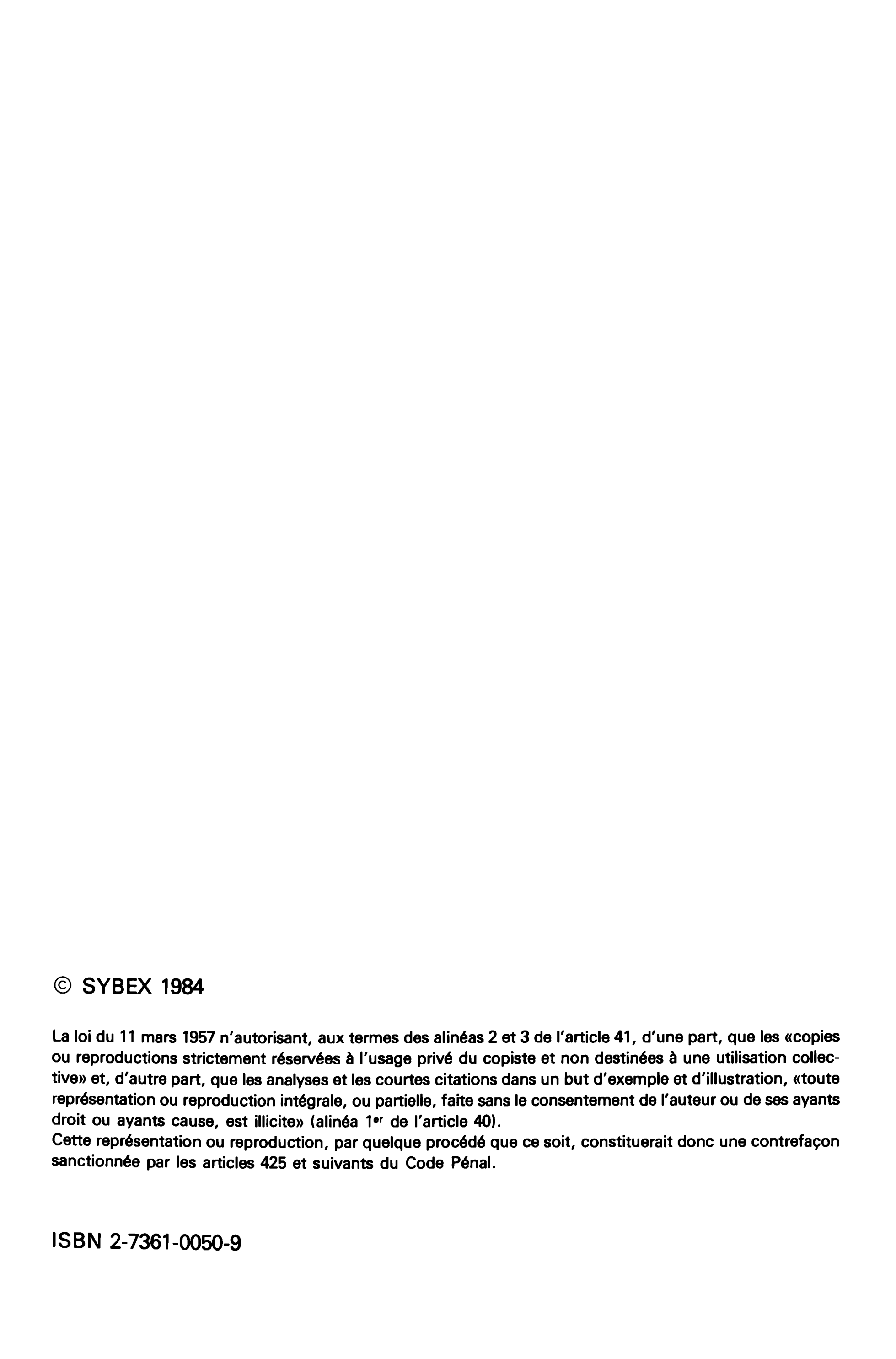
TO7

PROGRAMMATION EN ASSEMBLEUR

FAGOT-BARRALY



Paris • Berkeley • Düsseldorf • Londres



ISBN 2-7361-0050-9

INTRODUCTION

Ce livre est destiné à tous ceux qui connaissent les éléments de base de la programmation BASIC du TO 7 et qui souhaitent maintenant franchir l'étape suivante, l'étape du langage machine et de sa forme plus compréhensible : l'assembleur.

Il existe des ouvrages très bien faits qui traitent de ce sujet mais la plupart d'entre eux ne font qu'une place très réduite à la façon dont se connectent le BASIC et le langage machine. Aussi, pour notre part, nous nous sommes fixé la règle suivante : pour chacun des exemples étudiés nous ferons apparaître la partie assembleur, sa traduction en langage machine et la façon d'inclure ces codes machine dans un programme BASIC. Face à son ordinateur, le lecteur pourra donc, au fur et à mesure, mettre en pratique les connaissances qu'il viendra d'acquérir. Précisons que tous les exemples de ce livre ont été testés sur le TO 7 muni de la cartouche BASIC et sans aucune extension.

Le Chapitre 1 donnera les bases indispensables de l'arithmétique binaire car, ne l'oublions pas, un ordinateur ne connaît en réalité pas autre chose que les chiffres 0 et 1.

Le Chapitre 2 rappellera comment est conçue la mémoire écran du TO 7. Cette étude est rendue nécessaire par le fait que la majorité des programmes écrits en langage machine sont des animations de type vidéo. On trouvera aussi dans ce chapitre un programme de démonstration qui permettra de voir la différence flagrante dans les vitesses d'exécution d'un programme BASIC et de son équivalent assembleur.

Le Chapitre 3 nous fera pénétrer au cœur du microprocesseur : c'est un chapitre consacré aux différents registres, registres dont la connaissance est obligatoire pour aborder la suite de ce livre. Nous aborderons aussi dans ce chapitre l'étude des différentes façons d'utiliser une instruction assembleur suivant le mode d'adressage choisi.

Le Chapitre 4 sera consacré à l'étude de notre premier programme écrit en assembleur : les moindres détails seront expliqués.

Le Chapitre 5 analysera les principales instructions nécessaires à la programmation du microprocesseur du TO 7. De nombreux exemples seront fournis et ceci toujours avec la façon dont le BASIC et le langage machine seront reliés l'un à l'autre. Nous avons enchaîné l'étude des diverses instructions sans nous soucier d'un quelconque ordre logi-

que ou alphabétique : c'est la seule notion de progressivité qui nous a guidés.

Notre principal souhait est d'avoir fait un livre accessible facilement, un livre que l'on ne referme pas au bout de quelques pages devant la supposée trop grande ampleur de la difficulté.

1

L'ARITHMÉTIQUE BINAIRE

LES SYSTÈMES DE NUMÉRATION

1. La base dix

Le système de numération à base 10 est le système que nous utilisons dans la vie de tous les jours. On le connaît sous le nom de système décimal et c'est le nombre 10 qui y joue le rôle primordial.

Pour commencer notre étude rappelons ce que valent les premières puissances de 10 :

$10^0 = 1$
$10^1 = 10$
$10^2 = 10 \times 10 = 100$
$10^3 = 10 \times 10 \times 10 = 1000$

Par convention, n'importe quel nombre avec l'exposant 0 est égal à 1, et 10 n'échappe pas à la règle : $10^0 = 1$.

A l'aide de ces puissances, il est possible d'écrire un quelconque nombre entier.

$2548 = 2000 + 500 + 40 + 8$
Or $2000 = 2 \times 1000 = 2 \times 10^3$
$500 = 5 \times 100 = 5 \times 10^2$
$40 = 4 \times 10 = 4 \times 10^1$
$8 = 8 \times 1 = 8 \times 10^0$

Ce qui donne : $2548 = 2 \times 10^3 + 5 \times 10^2 + 4 \times 10^1 + 8 \times 10^0$.

D'une manière analogue, on aura :

$4706 = 4000 + 700 + 6$
Soit : $4706 = 4 \times 10^3 + 7 \times 10^2 + 0 \times 10^1 + 6 \times 10^0$.

$10^3 = 1000$	$10^2 = 100$	$10^1 = 10$	$10^0 = 1$
2	5	4	8
4	7	0	6

Naturellement, il nous est loisible de choisir des nombres plus grands car il suffira de prendre des puissances de 10 avec un exposant supérieur.

Rien de bien compliqué dans tout cela. Passons à l'étude d'une autre base mais, auparavant, notons bien quelque chose que nous retrouverons dans tout ce chapitre : les chiffres utilisés en base 10 vont de 0 à 9 ; ils sont tous inférieurs à cette base.

2. La base cinq

Les puissances de 5 se calculent facilement : $5^0 = 1$; $5^1 = 5$; $5^2 = 25$; $5^3 = 125$. Pour écrire un nombre en base 5, il va falloir constituer un tableau analogue au précédent mais, bien entendu, sa première ligne sera écrite avec les puissances de 5. Soit par exemple à traduire 138 dans le système à base 5 :

$5^3 = 125$	$5^2 = 25$	$5^1 = 5$	$5^0 = 1$
1	0	2	3

On a cherché combien de multiples de 125 (5^3) étaient contenus dans 138 :

1 fois et il reste 13 : $138 = 1 \times 125 + 13$.

Puis on a cherché combien de fois on pouvait faire rentrer 25 (5^2) dans 13 :

0 fois et il reste toujours 13 : $138 = 1 \times 125 + 0 \times 25 + 13$.

Il a fallu alors chercher combien de fois allait rentrer 5 (5^1) dans 13 :

2 fois et il reste 3 : $138 = 1 \times 125 + 0 \times 25 + 2 \times 5 + 3$.

Dernière phase de l'opération : dans le reste qui vaut à ce moment-là 3, combien de fois peut-on faire rentrer 1 (5^0) ?

3 fois et il ne reste rien : $138 = 1 \times 125 + 0 \times 25 + 2 \times 5 + 3 \times 1$.

On a donc en résumé :

$$138 = 1 \times 5^3 + 0 \times 5^2 + 2 \times 5^1 + 3 \times 5^0$$

et on en déduit que 138 s'écrit 1023 en base 5.

Prenons un deuxième exemple : quelle est la valeur de 279 en base 5 ?

$5^3 = 125$	$5^2 = 25$	$5^1 = 5$	$5^0 = 1$
2	1	0	4

$$279 = 2 \times 125 + 1 \times 25 + 0 \times 5 + 4 \times 1$$

ou $$279 = 2 \times 5^3 + 1 \times 5^2 + 0 \times 5^1 + 4 \times 5^0$$

Par suite 279 s'écrit 2104 en base 5.

En pratique, pour écrire un nombre décimal dans une autre base, on utilise le plus souvent la méthode dite «des divisions successives».

```
279 | 5
  4 | 55 | 5
       0 | 11 | 5
            1 |  2 | 5
                 2 | 0
```

Elle consiste à diviser le nombre par 5 puis le quotient par 5 puis le nouveau quotient obtenu par 5 et ceci jusqu'à ce que le dernier quotient soit nul. Il ne reste plus alors qu'à écrire la liste des différents restes en prenant la précaution essentielle de les copier dans l'ordre inverse. Les restes, dans notre exemple, étant 4,0,1,2 on écrit alors : 279 = 2104 (base 5).

Si nous avons maintenant à traduire en décimal un nombre déjà écrit en base 5, il faudra inscrire ce nombre dans un tableau conçu comme les précédents et ensuite le calculer.

Soit à écrire 3421 (base 5) en base 10.

$5^3 = 125$	$5^2 = 25$	$5^1 = 5$	$5^0 = 1$
3	4	2	1

On en déduit que 3421 (base 5) $= 3 \times 5^3 + 4 \times 5^2 + 2 \times 5^1 + 1 \times 5^0$.
Et l'on obtient : 3421 (base 5) $= 3 \times 125 + 4 \times 25 + 2 \times 5 + 1 \times 1 = 486$.

Remarquons, pour terminer, que les seuls chiffres utilisés en base 5 sont 0, 1, 2, 3, 4.

Il est conseillé au lecteur de s'assurer, avec quelques exercices dont il aura pris les nombres au hasard, que tout ce qui a été vu est bien assimilé. Non pas que la base 5 ait une quelconque importance en informatique, mais elle permet de comprendre sans peine les mécanismes des systèmes de numération.

3. La base deux

Nous arrivons maintenant au cœur du problème : voici le système de numération (dit système binaire) qu'utilisent les ordinateurs.

Tout d'abord, les puissances de 2 : $2^0 = 1$, $2^1 = 2$, $2^2 = 4$, $2^3 = 8$, $2^4 = 16$.

Puis maintenant, un exemple : on décide d'écrire 23 en binaire :

$2^4 = 16$	$2^3 = 8$	$2^2 = 4$	$2^1 = 2$	$2^0 = 1$
1	0	1	1	1

La plus grande puissance de deux qui rentre dans 23 est 16 (2^4) : le reste est 7. Peut-on ensuite faire rentrer 8 (2^3) dans 7 : la réponse est non et le chiffre 0 a été placé dans la case correspondante.

Par contre 4 (2^2) est contenu dans 7 : on écrit le chiffre 1 dans la troisième case et on note le nouveau reste : 3.

2 (2^1) étant plus petit que 3, on écrit le chiffre 1 dans la quatrième case et puisque le reste vaut alors 1, il nous faut encore écrire 1 mais cette fois-ci dans la dernière colonne.

23 = 10111 (base 2).

Heureusement pour nous, la méthode des divisions successives par 2 va nous donner la réponse d'une manière plus sûre et plus rapide :

```
23 | 2
 1 | 11 | 2
      1 |  5 | 2
           1 | 2 | 2
               0 | 1 | 2
                   1 | 0
```

23 = 10111 (2)

Voici d'autres exemples dont les calculs intermédiaires seront laissés à la charge du lecteur :

34 = 100010 (2)
150 = 10010110 (2)
255 = 11111111 (2)

Il reste à voir comment passer de la base 2 à la base 10.

Admettons que l'on veuille écrire 1111011 en décimal. On reconstitue le tableau dans lequel sont indiquées les puissances de 2 et on y écrit notre nombre :

$2^6 = 64$	$2^5 = 32$	$2^4 = 16$	$2^3 = 8$	$2^2 = 4$	$2^1 = 2$	$2^0 = 1$
1	1	1	1	0	1	1

On passe plus de temps à faire le tableau qu'à obtenir la réponse !

1111011 = 64 + 32 + 16 + 8 + 2 + 1 = 123 (décimal)

Il est nécessaire de remarquer que, dans ce que nous venons de voir, les seuls chiffres utilisés sont le 0 et le 1, à savoir les chiffres inférieurs à la base.

4. La base seize

C'est le système (appelé hexadécimal ou hexa) dont les informaticiens ne peuvent se passer, alors qu'au premier abord on pourrait se demander ce que vient faire son étude dans ce livre.

Les 16 chiffres nécessaires à l'écriture dans cette base sont tout d'abord 0,1,2,3,4,5,6,7,8,9 ...

Mais après le 9, le 10 peut-être ? Mais non, puisque c'est un nombre. Comme il nous manque 6 chiffres, on les a remplacés par les premières lettres de l'alphabet.

Chiffres	A	B	C	D	E	F
Valeurs	10	11	12	13	14	15

Ainsi 12 s'écrit C, 14 s'écrit E.

Là encore, les méthodes de conversion étudiées dans les paragraphes précédents vont s'appliquer.

Soit à écrire 300 en base 16 : les divisions successives doivent se faire par 16.

```
300 | 16
  C | 18 | 16
       2 |  1 | 16
            1 | 0
```

300 = 12C (hexa)

Passons à un autre exemple après avoir remarqué que le reste de la première division, qui valait 12, a été remplacé par C.

```
5032 | 16
   8 | 314 | 16
           A | 19 | 16
                3 |  1 | 16
                     1 | 0
```

5032 = 13A8 (hexa)

Si l'on souhaite traduire en décimal un nombre déjà écrit en base 16, on utilise les puissances de 16.

$16^0 = 1 \qquad 16^1 = 16 \qquad 16^2 = 256 \qquad 16^3 = 4096$

3D4F (hexa) s'écrit $3 \times 16^3 + D \times 16^2 + 4 \times 16^1 + F \times 16^0$
donc $3D4F = 3 \times 4096 + 13 \times 256 + 4 \times 16 + 15$
soit 3D4F = 15695 (base décimale).

Faut-il rappeler aux utilisateurs du TO 7 qu'il existe une fonction BASIC, HEX$, qui donne immédiatement la valeur hexadécimale d'un nombre décimal ?

PRINT HEX$ (15695) et l'ordinateur affichera 3D4F

Il nous faut maintenant comprendre où réside l'intérêt en informatique du système hexadécimal et pour cela comparer les représentations d'un même nombre décimal suivant que l'on veuille l'écrire en base 2 ou en base 16.

$$183 \text{ (décimal)} = \underbrace{1011}_{B}\ \underbrace{0111}_{7} \text{ (binaire)}$$

183 (décimal) = B 7 (hexa)

On sépare les huit chiffres binaires en deux groupes de quatre :

1011 et 0111

Or $1011 = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 8+2+1 = 11$ (décimal)

Et $0111 = 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 4+2+1 = 7$ (décimal)

En remarquant que 11 décimal s'écrit B en hexadécimal, on voit de façon immédiate la correspondance entre les bases 2 et 16. Il est tout à fait possible de passer directement de la base 2 à la base 16 sans avoir à connaître précisément le nombre décimal dont il s'agit.

Essayons encore en partant du nombre décimal 143 qui s'écrit 10001111 en base 2 :

$$\underbrace{1000}_{8}\ \ \underbrace{1111}_{F} = 8F \text{ en hexadécimal}$$

Bien entendu, on passera tout aussi facilement de la base 16 à la base 2 en ayant bien à l'esprit le tableau suivant.

Décimal	Binaire	Hexadécimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Que peut bien valoir par exemple en binaire le nombre hexadécimal 4A ?

Réponse : $\underbrace{0100}_{4}\ \underbrace{1010}_{A}$

Et le nombre hexadécimal 37E ?

Réponse : $\underbrace{0011}_{3}\ \underbrace{0111}_{7}\ \underbrace{1110}_{E}$

Dans ce dernier exemple, les deux premiers chiffres 0 sont inutiles et ne servent qu'à la compréhension de la règle qu'il faudra toujours respecter : le partage du nombre binaire doit se faire par groupes de quatre et ceci toujours en partant de la droite.

Puisque le TO 7 dispose de la fonction HEX$, il nous sera possible d'éviter la tâche fastidieuse de conversion d'un nombre en binaire et cela grâce à l'utilisation intermédiaire de la base 16.

Admettons que l'on veuille convertir 1000 (décimal) en binaire

PRINT HEX$ (1000) donne 3E8

On en déduit sans peine le résultat recherché :

$$\underbrace{0011}_{3}\ \underbrace{1110}_{E}\ \underbrace{1000}_{8}$$

OPÉRATIONS DANS LES BASES 2 ET 16

Nous nous limiterons dans ce paragraphe à l'addition et à la soustraction.

1. Addition

On considère le mécanisme de l'addition dans notre système décimal et on l'applique à la base 2.

```
  207
+ 321
-----
= 528
```

Dans cet exemple, les chiffres de chaque colonne s'ajoutent : comme on n'atteint jamais 10 (la base, ne pas l'oublier), il n'y a aucun problème. Il va en être de même dans les additions binaires suivantes car les totaux ne dépasseront jamais 2 (valeur de la base binaire) :

```
  101100          100011
+ 010001        + 000100
--------        --------
= 111101        = 100111
```

Reste à voir le cas de la retenue :

```
   1
  447
+ 223
-----
= 670
```

Dans cette addition décimale, 7 et 3 donnent 10, c'est-à-dire très précisément la valeur de la base. On écrit alors 0 au-dessous des chif-

fres 7 et 3 puis on reporte 1 dans la colonne suivante. Nous procéderons exactement de la même façon avec le système binaire.

```
     1
  10001
+ 01001
-------
= 11010
```

La somme des deux chiffres de droite donne 2 (valeur de la base). Le dernier chiffre du résultat sera donc un 0 et la retenue 1 sera écrite en haut de la colonne suivante. Le reste des calculs s'effectue ensuite sans difficulté.

Essayons encore :

```
    1 1
  100101
+ 000101
--------
= 101010
```

Il n'y a rien à redire, passons à un autre exemple :

```
     11
  101011
+ 010011
--------
= 111110
```

La somme des deux chiffres de droite donnant 2, on a écrit 0 comme dernier chiffre pour la réponse et on a retenu 1. L'addition de cette retenue avec les deux chiffres 1 de la deuxième colonne donne alors 3, ce qui se traduit par l'écriture du chiffre 1 dans la réponse et la pose d'une retenue en haut de la troisième colonne.

Il faut bien reconnaître que le risque d'erreur n'est pas négligeable lorsque l'on a à effectuer des calculs en binaire. Aussi, une méthode souvent utilisée consiste à traduire les nombres en hexadécimal, à les ajouter alors, puis à reconvertir si nécessaire le résultat en base 2.

Décidons de faire en hexadécimal les additions suivantes :

```
                      1
  34B5             5264
+ 6614             A32E
------           ------
= 9AC9           = F592
```

Si l'on se souvient de la correspondance :

A = 10 B = 11 C = 12 D = 13 E = 14 F = 15

on comprend directement comment la première opération a été faite.

5 + 4 = 9
B + 1 = 11 + 1 = C
4 + 6 = 10 = A
3 + 6 = 9

Pour ce qui concerne la deuxième addition, les choses se décomposent de la manière suivante :

4 + E = 4 + 14 = 18.

La retenue qui correspond à 10 dans notre système habituel est égale à 16 dans le système hexadécimal. Ce qui fait qu'après avoir posé la retenue en haut de la deuxième colonne, il restera 2 à écrire comme chiffre de droite de la réponse, réponse qui se complète ensuite par :

1 + 6 + 2 = 9
2 + 3 = 5
5 + A = 5 + 10 = 15 = F

Autres exemples :

```
   1 1                 111
   4BC3               FFFF
 + 2A2F             + FFFF
 -------            -------
 = 75F2             = 1FFFE
```

Nous sommes bien d'accord, n'est-ce pas, dans le système décimal il n'y a de retenue qu'à partir de 16.

2. Soustraction

Gardons le système précédent de numération et intéressons-nous au calcul d'une différence :

```
   9AE7
 - 49B3
 -------
 = 5134
```

C'est, somme toute, plutôt facile à comprendre :

7 − 3 = 4

E − B = 14 − 11 = 3
A − 9 = 10 − 9 = 1
9 − 4 = 5

Alors, essayons les retenues :

```
  9B54
- 6A29
------
= 312B
```

On a tendance à dire 9 ôté de 14, la force de l'habitude nous faisant rajouter une dizaine à 4. En réalité, puisque nous sommes en hexadécimal, ce n'est pas dix que l'on doit ajouter à 4 mais seize. Il s'agit, du coup, de faire 9 ôté de 20 : reste 11 c'est-à-dire B. Naturellement, la retenue ne doit pas être perdue dans la suite des calculs.

5 − 3 (dont 1 de retenue) = 2
B − A = 11 − 10 = 1
9 − 6 = 3

Deux autres exemples :

```
  4A85          ABCD
- 1F2E        - 2FFF
------        ------
= 2B57        = 7BCE
```

Les utilisateurs du TO 7 auront toutes les facilités pour se familiariser avec ce genre d'exercices. Pour faire vérifier par la machine ces deux calculs, il suffira de taper :

```
PRINT HEX$ (&H4A85 - &H1F2E)
```

et

```
PRINT HEX$ (&HABCD - &H2FFF)
```

On en arrive maintenant au calcul de la différence entre deux nombres écrits dans le système binaire. La méthode de soustraction directe peut être employée :

```
  101011
- 001001
--------
= 100010
```

Mais les programmeurs lui préfèrent une autre méthode, celle dite du «complément à deux», car on comprend bien que la soustraction

qui vient d'être effectuée aurait été plus compliquée si des retenues étaient apparues.

3. Le complément à deux

Considérons le nombre décimal 17.

Sa conversion en binaire donne 10001. Pour obtenir le complément à 2 de ce nombre, on respecte les trois étapes suivantes :

- on écrit notre nombre sur huit chiffres en rajoutant des 0 devant :
  00010001
- on remplace chaque 0 par 1 et chaque 1 par 0 :
  11101110
- on ajoute 00000001 à ce résultat :
  11101111

Le nombre que l'on obtient est appelé le complément à 2, sur huit chiffres, de 17, et l'ordinateur considérera que c'est l'opposé de 17, c'est-à-dire le nombre − 17. Oui, vous avez bien lu, dans le mode complément à 2, le nombre binaire 11101111 est égal à − 17 !

Comment s'en assurer ? En partant de l'idée toute simple qui consiste à dire : puisque, en ajoutant 17 et son opposé − 17, on obtient 0, on doit normalement, en ajoutant 00010001 et 11101111, obtenir aussi 0.

Voyons cela :

```
     11111111
      00010001
+     11101111
--------------
= (1)00000000
```

Les deux chiffres 1 de la droite font apparaître une retenue que l'on retrouve ensuite de colonne en colonne. Il faut tout de même noter qu'il ne doit pas être tenu compte de la dernière retenue et que nous prendrons l'habitude de la négliger. Nous verrons bientôt que l'ordinateur ne procède pas autrement : pour lui aussi, la dernière retenue de gauche tombe «à l'eau».

Un autre exemple : essayons d'écrire − 50 en binaire sous la forme complément à 2 :

00110010	50 décimal
11001101	chiffres inversés
11001110	ajout de 1

Donc −50 s'écrit 1100 1110 en binaire
ou C E en hexadécimal

Voici, tels quels, quelques résultats qui doivent permettre au lecteur d'assimiler parfaitement la façon dont l'ordinateur écrit les nombres négatifs :

−5 (décimal) = 11111011 (binaire) = FB (hexa)
−20 (décimal) = 11101100 (binaire) = EC (hexa)
−100 (décimal) = 10011100 (binaire) = 9C (hexa)

Avant de passer à autre chose, revenons quelques minutes sur la façon dont on s'y prendra pour faire une différence binaire maintenant que nous savons utiliser la technique du complément à 2.

Soit à calculer 101000 − 10111.

On cherche l'opposé du deuxième terme de la soustraction en mode complément à 2 : on obtient 11101001.

Il reste alors à ajouter le premier terme avec l'opposé du deuxième :

```
   111 1
    00101000
+   11101001
------------
= (1)00010001
```

La réponse est la suivante : 101000 − 10111 = 10001.

OPÉRATEURS LOGIQUES

En dehors des calculs arithmétiques habituels, on peut effectuer sur les nombres binaires des opérations d'un type spécial que l'on appelle les opérations logiques. Elles ne présentent aucune difficulté car en aucun cas ne se pose le problème des retenues.

1. Le OU logique

Cette opération respecte les règles suivantes :

```
     0          0          1          1
  OU 0       OU 1       OU 0       OU 1
------     ------     ------     ------
   = 0        = 1        = 1        = 1
```

C'est la même chose avec des nombres binaires plus grands :

```
   101101                  101000
OU 110101               OU 001100
---------               ---------
 = 111101                = 101100
```

Le TO 7 dispose d'une instruction qui effectue ce type de calculs : c'est le mot clé OR. Demandons-lui quelques résultats :

PRINT 46 OR 100 ; réponse : 110

```
    101110       46
OR 1100100       100
----------
=  1101110       110
```

PRINT 50 OR 0 ; réponse : 50

```
   110010       50
OR 000000       0
---------
=  110010       50
```

L'opérateur OR va nous servir en assembleur car il permet de forcer l'un des chiffres binaires à passer à 1. Voyons comment :

PRINT 82 OR 1 ; réponse : 83

```
   1010010       82
OR 0000001       1
----------
=  1010011       83
```

PRINT 91 OR 1 ; réponse : 91

```
   1011011       91
OR 0000001       1
----------
=  1011011       91
```

Dans le premier exemple, on part d'un nombre dont le dernier chiffre binaire (bit 0) est égal à 0. Après utilisation de OR 1, ce dernier chiffre a été porté à 1 sans qu'aucun des autres chiffres ait été modifié.

Dans le deuxième cas, on est parti d'un nombre qui se terminait déjà par un 1. OR 1 n'a modifié ni ce chiffre ni naturellement aucun des autres. On en conclut donc que si l'on effectue OR 1 avec n'importe quel nombre, on aura un résultat dont le dernier chiffre (bit 0) vaudra obligatoirement 1.

D'une façon analogue, en calculant OR 4 avec n'importe quel nombre, on sera certain que le troisième chiffre en partant de la droite est un 1 (bit 2) :

PRINT 19 OR 4 ; réponse : 23

```
     10011 <------- 19
OR   00100 <------- 4
----------
=    10111 <------- 23
```

Le troisième chiffre est bien passé à 1.

PRINT 52 OR 4 ; réponse : 52

```
     110100 <------- 52
OR   000100 <------- 4
-----------
=    110100 <------- 52
```

Le troisième chiffre est resté à 1.

2. Le ET logique

Le ET logique est défini par les règles suivantes :

```
   0        0        1        1
ET 0     ET 1     ET 0     ET 1
----     ----     ----     ----
 = 0      = 0      = 0      = 1
```

Quelques exemples :

```
    101100          101000
 ET 011001       ET 110111
 ---------       ---------
  = 001000        = 100000
```

On peut faire faire ces calculs par l'ordinateur et cette fois, c'est le mot réservé AND qui va nous servir.

PRINT 30 AND 40 ; réponse : 8

```
      11110 <------- 30
AND  101000 <------- 40
-----------
=    001000 <------- 8
```

On retrouve l'instruction AND en assembleur car, grâce à elle, nous pouvons mettre à 0 n'importe quel chiffre binaire. Supposons que nous

ayons un nombre et que nous voulions forcer à 0 son chiffre de droite (bit 0). On utilisera AND 254 et voici pourquoi :

```
      PRINT 201 AND 254 ; réponse : 200
          11001001 ←——— 201
      AND 11111110 ←——— 254
      ----------------
      =   11001000 ←——— 200
```

Seul le dernier chiffre a été mis à 0, les autres sont restés les mêmes. 254 a en effet la particularité d'être constitué de sept chiffres 1 suivis d'un seul 0.

Si nous étions partis d'un nombre se terminant déjà par 0, AND 254 n'aurait rien modifié, ce qui nous permet de donner la conclusion suivante : quel que soit le nombre considéré, en le combinant avec 254 on pourra être assuré qu'il se terminera par 0.

Il est possible d'annuler n'importe quel chiffre d'un nombre avec l'opérateur AND. AND 124, par exemple, annulera le chiffre de gauche (bit 7) mais en même temps les deux chiffres de droite (bits 0 et 1) de n'importe quel nombre de huit chiffres.

```
      PRINT 245 AND 124 ; réponse : 116
          11110101 ←——— 245
      AND 01111100 ←——— 124
      ----------------
      =   01110100 ←——— 116
```

3. Le OU exclusif logique

Noté XOR, le OU exclusif obéit aux mêmes règles que le OU déjà défini sauf pour la quatrième partie :

```
      0          0          1          1
  XOR 0      XOR 1      XOR 0      XOR 1
  -----      -----      -----      -----
    = 0        = 1        = 1        = 0
```

Le résultat n'est égal à 1 que lorsqu'un des chiffres et un seulement est égal à 1.

Tapons au clavier de notre ordinateur :

```
      PRINT 30 XOR 40 ; réponse : 54
           11110 ←——— 30
      XOR 101000 ←——— 40
      --------------
      =   110110 ←——— 54
```

```
PRINT 25 XOR 100 ; réponse : 125
        11001 ←——— 25
XOR 1100100 ←——— 100
-----------
=     1111101 ←——— 125
```

L'opérateur XOR est mis en œuvre à chaque fois que l'on veut faire passer à 1 les chiffres 0 et à 0 les chiffres 1. Supposons que l'on ait un nombre dont on veuille faire changer d'état le dernier chiffre (bit 0) : on le combinera avec XOR 1. Si le nombre se terminait par 0, il se terminera alors par 1 mais par contre, si son dernier chiffre était 1, ce sera du coup 0. On essaie :

```
PRINT 28 XOR 1 ; réponse : 29
      11100 ←——— 28
XOR 00001 ←——— 1
---------
=     11101 ←——— 29

PRINT 31 XOR 1 ; réponse : 30
      11111 ←——— 31
XOR 00001 ←——— 1
---------
=     11110 ←——— 30
```

Bien entendu, XOR peut être utilisé pour faire basculer d'un état à l'autre n'importe lequel des chiffres sans modifier les autres. Par exemple, XOR 5 ne changera les états que du premier et du troisième chiffres en partant de la droite (5 est égal à 101 en binaire).

2

LA MÉMOIRE ÉCRAN

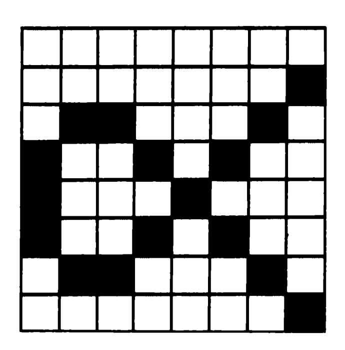
Le TO 7 transmet au téléviseur une image dont les caractéristiques sont données par l'instruction SCREEN C,F,P.

C est la couleur des caractères, points lignes et graphiques qui apparaîtront quand le programme sera exécuté. F est la couleur de fond de la partie centrale de l'écran qui constitue la fenêtre de travail et P est la couleur de pourtour de l'écran. Le programmeur n'a pas accès à ce cadre extérieur et ne peut rien faire d'autre qu'en modifier la couleur.

LES CARACTÈRES GRAPHIQUES GR$

La possibilité est offerte aux utilisateurs du TO 7 de définir de nouveaux caractères à condition d'en avoir fait auparavant la déclaration à l'aide de l'instruction CLEAR,,N. N est le nombre de GR$ que l'on va alors obtenir au moyen de l'expression DEFGR$. Pour cela on dessine chaque caractère nouveau dans un carré de 8 cases sur 8.

Voici, à titre d'exemple, la définition du symbole α. L'analyse qui va suivre permettra d'aborder facilement le reste du chapitre.



Première ligne
Deuxième ligne
Troisième ligne

Huitième ligne

Le caractère est inscrit dans un carré de 64 cases. On y a noirci celles qui feront ressortir la lettre α et on va demander à l'ordinateur de n'allumer que ces cases-là. Il serait naturellement fastidieux de prendre les cases les unes à la suite des autres et de préciser à chaque fois

si elles doivent être allumées ou éteintes. Aussi la définition d'un caractère se fait-elle ligne par ligne, en ayant à l'esprit que la convention suivante a été décidée une fois pour toutes : chaque case allumée sera représentée par le chiffre 1 et chaque case éteinte par le chiffre 0.

Ainsi, les 8 cases de la première ligne seront notées 00000000, les 8 cases de la deuxième ligne seront notées 00000001, celles de la troisième 01100010 etc. La forme finale de la définition de ce caractère sera donc :

```
DEFGR$(0)=&B00000000,&B00000001,&B01100010,&B10010100,
&B10001000,&B10010100,&B01100010,&B00000001
```

Le symbole &B placé devant chaque liste de 0 et de 1 indique qu'il s'agit d'un nombre binaire. L'étude du Chapitre 1 a fait comprendre au lecteur pour quelle raison nous sommes autorisés à écrire plus simplement :

```
DEFGR$(0)=0,1,98,148,144,148,98,1
```

Il faut retenir de ce paragraphe que chaque ligne est un segment de 8 cases que l'on peut allumer ou éteindre à volonté.

LA MÉMOIRE ÉCRAN DU TO 7

Sur la partie utilisable de l'écran de télévision, on peut faire entrer 40 caractères en largeur et 25 caractères en hauteur, ce qui fait que l'on peut écrire au total 40 × 25 soit 1000 caractères. Puisque le paragraphe précédent a montré que chaque caractère était constitué de 8 segments, on arrive au résultat suivant : l'image fournie par l'ordinateur est formée de 8000 segments, eux-mêmes formés de 8 cases ou points élémentaires. Il nous faut voir maintenant comment le programmeur va avoir la possibilité d'agir sur ces segments. Dans un premier temps c'est l'instruction POKE qui sera employée puis, très vite, viendra le moment d'expérimenter vos connaissances toutes neuves de l'assembleur.

Les 8000 segments sont numérotés de 0 à 7999. Il faut s'y faire, la logique en informatique veut qu'on commence toujours à 0, jamais à 1. A chacun de ces segments correspond dans la mémoire centrale

un octet, c'est-à-dire un ensemble de 8 bits numérotés là encore à partir de 0. Aussi, pour agir sur un segment il faudra connaître l'adresse de l'octet correspondant et écrire dans cet octet, avec POKE, l'information voulue. Si l'on veut par exemple allumer tous les points du 900e segment, on doit procéder de la façon suivante :

• Traduction de l'effet recherché en chiffres : comme on veut que les 8 points du segment soient allumés on obtient, de façon analogue à la définition d'une ligne de GR$, les 8 chiffres binaires 11111111 (soit 255 en décimal).

• Recherche de l'adresse A de l'octet qui correspond au segment numéro 900.

• Ecriture dans cet octet du nombre 255:POKE A,255.

Il faut comprendre dès lors la relation qu'il y a entre le numéro d'un segment et le numéro (ou adresse) de l'octet correspondant dans la mémoire. Le premier octet concerné a pour adresse 16384 (il correspond au segment numéro 0) et le dernier a pour adresse 24383 (il correspond au segment numéroté 7999). Aussi dans l'exemple précédent on aurait écrit : POKE 17284,255 puisque 17284 = 16384 + 900.

Voici maintenant les explications qui permettent de voir le rapport qu'il y a entre le numéro d'un segment et sa position précise sur l'écran : l'ordinateur considère que les segments sont alignés en fonction de l'ordre croissant de leurs numéros, c'est-à-dire les uns à la suite des autres et de gauche à droite.

Ainsi le segment 0 sera le premier segment en haut et à gauche, le segment 1 sera placé à droite, juste après le segment 0, le segment 2 sera à droite du segment 1. Comme un segment fait exactement la largeur d'un caractère, on peut donc placer exactement 40 segments sur la première ligne de l'écran (segments numérotés de 0 à 39).

Le segment 40 se retrouve à la ligne suivante au-dessous du segment 0, le 41 au-dessous du 1, le 80 au-dessous du 40, etc.

Afin de vous assurer que tout ceci est bien assimilé, essayez de prévoir ce qui va apparaître sur votre écran si vous faites exécuter la ligne suivante :

```
10 CLS: FOR I = 0 TO 39 : POKE 16384+I+J,255: J=J+40 : NEXT
```

Voyons, toujours à titre d'exemple, comment utiliser ce qui vient d'être établi pour dessiner, à l'aide de l'instruction POKE, la lettre

α au milieu de l'écran. On choisit comme octet correspondant au segment du haut de ce caractère, l'octet 20000. Il faut écrire dans cet octet la valeur 0 car la première ligne est constituée de cases vides.

```
POKE 20000,0
```

La ligne 2 dans laquelle il faudra allumer le point de droite correspond à l'octet 20000 + 40

```
POKE 20040,1
```

En augmentant les adresses des octets par pas de 40 et en y inscrivant à chaque fois les valeurs convenables, on arrivera au dessin complet de la lettre α sur l'écran. Le programme qui suit retrace dans une boucle les 8 opérations nécessaires.

```
10 FOR I = 20000 TO 20280 STEP 40
20 READ J : POKE I,J : NEXT
30 DATA 0,1,98,148,144,148,98,1
```

Il reste à donner la formule qui servira à allumer un segment dont on connaît l'emplacement sur l'écran. Si C est sa colonne (de 0 à 39) et L sa ligne (de 0 à 199), on écrira avec POKE dans l'octet ayant pour adresse :

```
16384 + 40 * L + C
```

LES COULEURS DU TO 7

Rappelons que le système TO 7 dispose de 8 couleurs codées de 0 à 7 s'il s'agit des couleurs des caractères et de – 1 à – 8 s'il s'agit des couleurs des fonds sur lesquels sont dessinés ces caractères.

Sur votre ordinateur, des contraintes techniques font que vous ne pourrez jamais colorier un segment de 8 points à l'aide de plus de 2 couleurs.

Couleur	Code caractère	Code fond
NOIR	0	−1
ROUGE	1	−2
VERT	2	−3
JAUNE	3	−4
BLEU	4	−5
VIOLET	5	−6
BLEU CLAIR	6	−7
BLANC	7	−8

L'instruction BASIC qui détermine la couleur d'un caractère et celle de son fond est COLOR. Si l'on désire par exemple, qu'un caractère apparaisse en rouge sur fond jaune on se servira de la forme COLOR 1,3.

Cette manière très simple de procéder pour choisir ses couleurs ne sera pas utilisable dès que l'étude des premiers programmes assembleurs sera commencée. Aussi faut-il chercher une autre méthode et s'intéresser de près à l'octet 59331 et plus précisément au bit le plus à droite de cet octet (bit 0). C'est en effet lorsque ce bit vaudra 0 que nous allons pouvoir par POKE déterminer les couleurs.

Voyons les détails : l'octet 59331 est naturellement constitué de 8 bits mais 7 de ceux-ci sont absolument étrangers à notre affaire. Tout notre travail va consister à agir sur le bit de droite sans bien sûr modifier les 7 autres bits.

Premier cas : écriture du chiffre 1 dans le bit 0.

Dans le chapitre 1 l'opérateur logique OR a été rencontré et l'un des exercices a montré comment forcer un des bits à passer à 1. Il suffit d'appliquer ici cette méthode en ne perdant pas de vue que seul le bit de droite doit être concerné.

```
X = PEEK(59331) OR 1 : POKE 59331,X
```

Pour comprendre mieux admettons que la valeur inscrite dans l'octet 59331 soit au début égale à 10110010 ; de ce fait et puisque la fonction PEEK nous donne le contenu de l'octet 59331, nous aurons :

```
PEEK (59331) = 10110010
```

Pour obtenir X il faudra alors effectuer un OU logique entre 10110010 et 1

```
        10110010 ←——— valeur initiale
OU      00000001 ←——— valeur 1
——————————————
X =     10110011 ←——— valeur finale
```

Il reste à écrire par un POKE la valeur finale dans l'octet voulu : le but cherché a été atteint puisque maintenant l'octet 59331 contient une valeur dont le bit de droite vaut 1 sans que les 7 autres bits aient été modifiés.

Le lecteur vérifiera facilement que si, dans l'exemple, un nombre binaire se terminant par 1 avait été choisi, la méthode aurait là encore fourni le résultat recherché (le bit de droite restant alors à 1).

Deuxième cas : écriture du chiffre 0 dans le bit de droite (bit 0).

On reprend ce qui a été dit au chapitre précédent concernant le ET logique et on l'applique à notre problème.

```
X = PEEK(59331) AND 254 : POKE 59331,X
```

En supposant que PEEK(59331) soit égal cette fois à 10011101 et en tenant compte du fait que 254 s'écrit 11111110 en binaire nous aurons :

```
        10011101 ←——— valeur initiale
AND     11111110 ←——— valeur 254
——————————————
X =     10011100 ←——— valeur finale
```

Quand on aura écrit la valeur X dans l'octet 59331 grâce à l'instruction POKE, on aura l'assurance que le bit 0 sera à la valeur 0 sans que le reste de l'octet ait subi un quelconque changement.

Dans ce deuxième cas aussi, il est facile de vérifier que la méthode aurait donné un résultat correct si l'on avait choisi, à titre d'exemple, un nombre se terminant par 0 (le bit 0 restant alors au niveau 0).

DÉTERMINATION DE LA COULEUR D'UN SEGMENT

Il faut maintenant en venir au fait et comprendre le petit mécanisme qui préside au choix de la couleur des points d'un segment.

Rien ne remplaçant un exemple, voici 3 lignes BASIC que nous allons étudier.

```
10 X = PEEK(59331) OR 1 : POKE 59331,X
20 POKE 16384,170
30 FOR I = 0 TO 7 : PRINT POINT(I,0); :NEXT
```

La ligne 10 reprend ce qui a été vu quelques lignes au-dessus : le bit le plus à droite (bit 0) de l'octet 59331 vaut 1.

La ligne 20 donne la forme du segment en écrivant 170 dans l'octet 16384 : puisque 16384 correspond au premier segment en haut à gauche de l'écran et que 170 s'écrit 10101010 en binaire, le segment s'allume en pointillés. Les couleurs de ce segment (bleu foncé sur bleu clair) sont les couleurs standard et la ligne 30 n'est là que pour confirmation : elle nous indique bien que le premier point est bleu foncé (couleur 4), le deuxième bleu clair (couleur − 7 car couleur de fond), le troisième bleu foncé, etc.

La suite du programme maintenant :

```
40 X = PEEK(59331) AND 254 : POKE 59331,X
50 POKE 16384,8
60 FOR I = 0 TO 7 : PRINT POINT(I,0); :NEXT
```

La ligne 40 place 0 dans le dernier bit de l'octet 59331.

La ligne 50, relative au premier segment de l'écran, ne concerne alors plus sa forme (qui restera en pointillés) mais sa couleur : le segment s'est allumé en rouge sur fond noir et la ligne 60 ici encore nous le confirme : le premier point est rouge (couleur 1), le deuxième est noir (couleur de fond − 1), le troisième est rouge, le huitième est noir. Le tableau suivant permet de choisir n'importe quelle combinaison de deux couleurs.

COULEURS

	0	1	2	3	4	5	6	7
NOIR	0	8	16	24	32	40	48	56
ROUGE	1	9	17	25	33	41	49	57
VERT	2	10	18	26	34	42	50	58
JAUNE	3	11	19	27	35	43	51	59
BLEU	4	12	20	28	36	44	52	60
VIOLET	5	13	21	29	37	45	53	61
BLEU CLAIR	6	14	22	30	38	46	54	62
BLANC	7	15	23	31	39	47	55	63

Voici quelques couples de couleurs obtenues en modifiant la ligne 50

Bleu foncé sur vert : POKE 16384,34
Blanc sur violet : POKE 16384,61
Vert sur vert : POKE 16384,18

Résumons les étapes à respecter lorsque l'on veut allumer l'un des 8000 segments de l'écran avec les couleurs de son choix : on met à 1 le bit 0 de l'octet 59331, on écrit dans l'octet correspondant au segment la valeur relative à la forme, on force le bit droit de l'octet 59331 à s'annuler et on écrit alors dans l'octet du segment la valeur relative à la couleur.

PROGRAMME DE DÉMONSTRATION

Pour clore ce chapitre, il est proposé une comparaison entre les vitesses d'exécution du même programme, suivant que celui-ci a été écrit en BASIC ou en assembleur. Gageons que la comparaison ne tournera pas à l'avantage de la version BASIC.

Le but des lignes qui suivent est d'allumer les 8000 segments de l'écran en utilisant à chaque fois l'une des sept premières couleurs.

```
 10 CLS : FOR I = 16384 TO 24383
 20 X = PEEK(59331) OR 1 : POKE 59331,X
 30 POKE I,255
 40 X = PEEK(59331) AND 254 : POKE 59331,X
 50 POKE I,C
 60 C = C+8 : IF C = 56 THEN C = 0
 70 NEXT
 80 CLS : LOCATE 0,15,0
 90 PRINT "VOICI MAINTENANT LA VERSION ASSEMBLEUR"
100 PLAY "L96PPP" ; CLEAR,32000
110 FOR I = 32001 TO 32043 :READ I$
120 POKE I, VAL("&H"+I$): NEXT
130 CLS : EXEC 32001
140 DATA C6,FF,8E,40,00,4F,34,02,B6,E7,C3,8A,01
150 DATA B7,E7,C3,E7,84,84,FE,B7,E7,C3,35,02,A7,80
160 DATA 8C,5F,40,27,OA,8B,08,81,38,27,DF,34,02,20,DE,39
```

La partie qui correspond à la version BASIC (lignes 0 à 80) se comprend aisément et est la mise en application de ce qui a été vu dans les paragraphes précédents : on met le bit de droite (bit 0) de l'octet 59331 à 1 (ligne 20) puis on écrit dans l'octet écran I la valeur 255 (ligne 30). Le segment aura donc la forme d'un trait plein de 8 points puisque 255 vaut 11111111 en binaire.

On met ensuite le bit 0 de l'octet 59331 à 0 (ligne 40) et on écrit dans l'octet I la valeur C (ligne 50) : le segment sera colorié en fonction de cette valeur C qui vaut successivement 0,8,16,24,32,40,48,0,8,16... Ainsi les petits segments seront de couleur noire, rouge, verte, jaune... sur fond noir, fond qui n'a d'ailleurs aucune espèce d'importance du fait que tous les segments de cet exemple ont leurs 8 points allumés.

Les explications permettant de comprendre la partie langage machine de ce programme sont données maintenant. Inutile de vous y intéresser, vous risqueriez de vous décourager devant ce que vous auriez tendance à considérer comme étant d'une difficulté insurmontable. Passez donc au chapitre suivant et soyez assurés que dans quelques jours ce qui suit vous paraîtra limpide.

Nombre d'octets : 43

Ligne	Codes machine	Assembleur		
1	C6 FF		LDB	#255
2	8E 40 00		LDX	#16384
3	4F	NOIR	CLRA	
4	34 02		PSHS	A
5	B6 E7 C3	SUITE	LDA	> 59331
6	8A 01		ORA	#1
7	B7 E7 C3		STA	> 59331
8	E7 84		STB	,X
9	84 FE		ANDA	#254
10	B7 E7 C3		STA	> 59331
11	35 02		PULS	A
12	A7 80		STA	,X+
13	8C 5F 40		CMPX	#24384
14	27 0A		BEQ	FIN(+10)
15	8B 08		ADDA	#8
16	81 38		CMPA	#56
17	27 DF		BEQ	NOIR(−33)
18	34 02		PSHS	A
19	20 DE		BRA	SUITE (−34)
20	39	FIN	RTS	

Ligne 1 : B contient la valeur 255 et gardera cette valeur durant tout le programme.

Ligne 2 : X est chargé avec le nombre 16384 qui est l'adresse du premier octet vidéo.

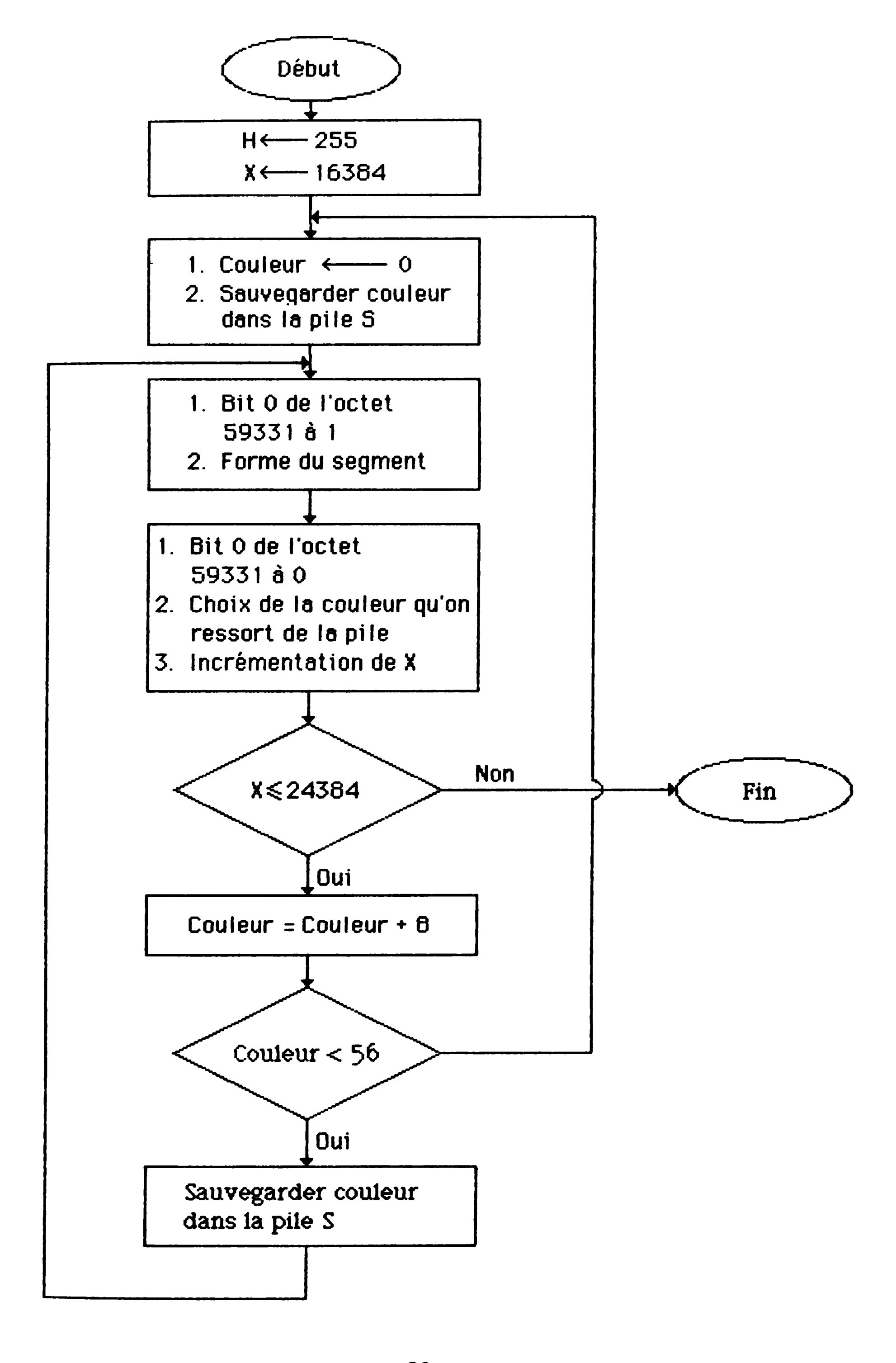
Ligne 3 : le registre A est mis à zéro (couleur noire).

Ligne 4 : on met de côté dans la pile S (on dit sauvegarde) la valeur contenue dans A et ceci tout simplement parce que A va être utilisé pour faire autre chose dans la partie suivante du programme.

Lignes 5, 6 et 7 : passage à 1 du bit 0 de l'octet 59331.

Ligne 8 : on allume les 8 points du segment dont l'adresse est dans le registre X. Au premier passage de la boucle il s'agit donc du segment 16384.

Lignes 9 et 10 : mise à zéro du bit 0 de l'octet 59331.


Début
H ⟵ 255
X ⟵ 16384
1. Couleur ⟵ 0
2. Sauvegarder couleur dans la pile S
1. Bit 0 de l'octet 59331 à 1
2. Forme du segment
1. Bit 0 de l'octet 59331 à 0
2. Choix de la couleur qu'on ressort de la pile
3. Incrémentation de X
X ≤ 24384
Non
Fin
Oui
Couleur = Couleur + 8
Couleur < 56
Oui
Sauvegarder couleur dans la pile S

Lignes 11 et 12 : on ressort de la pile S la valeur de la couleur que l'on écrit dans l'octet pointé par X. A la première lecture de la boucle le segment 16384 sera de couleur noire. Puis le registre X est incrémenté et contient alors l'adresse du segment suivant.

Lignes 13 et 14 : on compare X avec 24384 et s'il y a égalité, alors le branchement se fait sur la dernière ligne du programme ; les 8000 segments auront alors été allumés.

Lignes 15 et 16 : on ajoute 8 à l'accumulateur A et on regarde si l'on atteint ou non 56.

Ligne 17 : si c'est oui alors on se branche à la ligne 3 pour que la couleur reparte à 0.

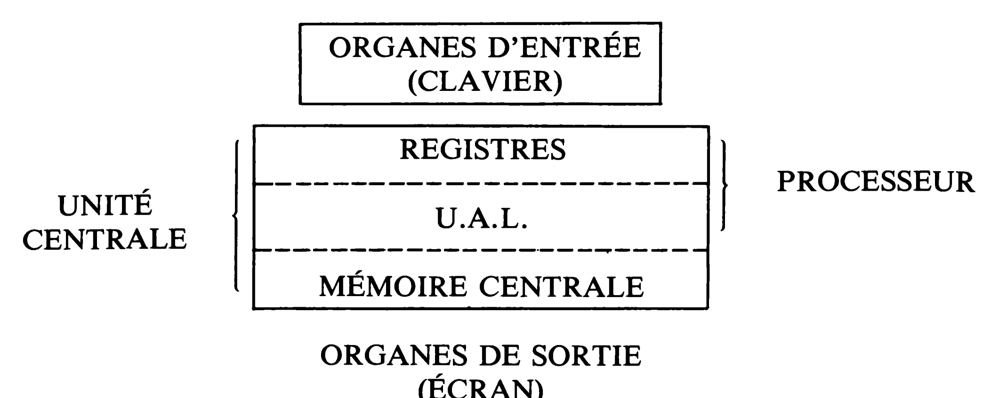
Lignes 18 et 19 : si c'est non, on remet de côté dans la pile la nouvelle valeur de la couleur et on repart à la ligne 5 pour s'occuper de l'octet 59331.

Voici l'organigramme correspondant au programme assembleur (voir ci-contre) :

3

L'ARCHITECTURE INTERNE DU MICROPROCESSEUR 6809

Avant de commencer l'étude du microprocesseur, il est nécessaire de le situer dans son environnement : aussi ce chapitre débutera par quelques rappels sur la structure générale d'un micro-ordinateur.



ORGANES DE SORTIE
(ÉCRAN)

C'est essentiellement la mémoire centrale qui sera utile à notre étude et c'est pour cette raison que les mémoires auxiliaires n'apparaissent pas sur le schéma.

LA MÉMOIRE CENTRALE

La mémoire centrale permet d'enregistrer, de conserver, et de restituer à la demande les informations qui lui ont été communiquées. Ces informations sont de deux sortes : ce sont soit des données soit des programmes. A priori, rien ne distingue en mémoire ces deux types d'informations et c'est la seule logique du programme qui empêchera la confusion.

Pour des raisons technologiques, l'information rangée en mémoire se trouve sous la forme de combinaisons des chiffres binaires 0 et 1. Le bit (ou chiffre binaire) est l'unité élémentaire d'information et ne peut prendre que l'une de ces valeurs. Un octet est constitué de huit bits. Le nombre le plus grand que l'on puisse avoir dans un octet est le nombre 11111111 (soit 255 en décimal). Le nombre le plus petit est obtenu quand les huit bits valent 0 : c'est le nombre 00000000 (ou 0 en décimal).

La mémoire centrale de la plupart des micro-ordinateurs est divisée en 65536 octets et l'ordinateur est capable de retrouver le contenu de

n'importe quel octet grâce à son adresse. Les octets de la mémoire sont numérotés de 0 à 65535 et c'est ce numéro que l'on appelle l'adresse. Il est possible de savoir le nombre que contient un octet avec la fonction BASIC PEEK. Essayons :

```
PRINT PEEK (10000) ; réponse : 38
```

Puisque 38 = 100110 en binaire, on peut retrouver la configuration exacte de l'octet numéro 10000 :

0	0	1	0	0	1	1	0
bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0

On peut dire aussi que cet octet contient le nombre hexadécimal 26.

Puisqu'on est capable de connaître la valeur qui se trouve dans un octet, on doit être capable de modifier cette valeur : l'instruction POKE est à notre disposition pour cela.

```
PRINT PEEK (30000) ; réponse : 255
POKE 30000, 100
PRINT PEEK (30000) ; réponse : 100
```

Avant notre intervention, l'octet numéro 30000 contenait le nombre 255. Tous les octets contiennent une valeur et il est difficile de dire, dans l'absolu, à quoi elle correspond. Nous avons inscrit par POKE la valeur 100 dans l'octet et il ne nous est plus resté qu'à en demander la confirmation avec PRINT PEEK (30000).

Un nouvel essai :

```
PRINT PEEK (5000) ; réponse : 4
POKE 5000, 100
PRINT PEEK (5000) ; réponse : 4
```

Eh oui, la commande POKE ne nous permet pas de modifier le contenu de n'importe quel octet de la mémoire ! Nous allons savoir pourquoi dès que nous aurons distingué les deux grands types de mémoire.

La ROM (*Read Only Memory*) est la partie de la mémoire centrale que l'on peut seulement lire. Il n'est pas question de modifier un octet qui se trouve dans cette zone-là. L'octet 5000 par exemple se trouve

	128	64	32	16	8	4	2	1
A	0	1	0	1	0	0	0	1

Dans A se trouve donc le nombre décimal 81 ou hexadécimal 51.

	128	64	32	16	8	4	2	1
B	1	0	0	0	1	1	0	0

Dans B c'est le nombre 140 (décimal) ou 8C (hexa) qui se trouve.

Pour former D, on associe A et B :

A								B							
0	1	0	1	0	0	0	1	1	0	0	0	1	1	0	0

La valeur de D est alors 0101000110001100 c'est-à-dire 20876 en décimal. Il faudra toujours se souvenir que D est un registre 16 bits et qu'il correspond donc à deux octets (soit un nombre compris entre 0 et 65535).

2. Les registres X et Y

Ce sont des registres 16 bits et donc on peut y écrire n'importe quel nombre de 0 à 1111111111111111 (16 fois le chiffre 1). Si nous prenons la peine de traduire cette valeur binaire en décimal, on obtient 65535. Ceci nous permet de comprendre le rôle que joueront X et Y : ils contiendront généralement une adresse mémoire et cette adresse pourra être celle de n'importe quel octet de l'intervalle 0 – 65535.

X et Y sont souvent appelés registres d'adresses eu égard à ce que nous venons de dire, mais on les appelle aussi registres d'index car on peut les utiliser dans le mode d'adressage indexé (nous verrons cela bientôt).

3. Les registres U et S

Ils portent le nom de pointeurs de piles : pile utilisateur pour U et pile système pour S. Notre étude va porter sur U, mais aurait tout aussi bien pu se faire avec S, à quelques détails près qui seront mentionnés ultérieurement.

Une pile est un endroit de la mémoire où seront stockés — empilés — des nombres les uns à la suite des autres. La structure de la pile d'un ordinateur correspond tout à fait à celle d'une pile d'assiettes : on peut

toujours rajouter une assiette sur la pile, mais si l'on veut en reprendre une, ce sera toujours la dernière posée que l'on devra récupérer (dernier entré, premier sorti).

L'utilisation de la pile n'étant pas évidente pour le programmeur qui fait ses premiers pas en assembleur, passons un peu de temps sur un exemple. Supposons que dans le processeur, les registres A, B, X et U aient les valeurs suivantes :

A	0	0	0	1	0	1	0	0

A contient donc 20 (décimal) ou 14 (hexadécimal)

B	0	0	1	0	0	0	1	1

B contient donc 35 (décimal) ou 23 (hexadécimal)

X	0	0	0	0	1	1	1	1	1	0	0	0	0	0	0	1

Dans le registre X est écrit le nombre décimal 3969 (0F81 hexadécimal)

U	0	1	1	1	1	1	1	1	1	0	1	1	1	1	0	0

Dans U se trouve le nombre 32700 (7FBC en hexadécimal).

La valeur 32700 indique dans cet exemple que nous allons empiler nos nombres dans une série d'octets de la mémoire à partir justement de l'octet 32700. Cet octet en lui-même ne sera pas concerné par notre travail car c'est dans l'octet juste à côté que va démarrer notre empilement.

PSHU A

Nous voilà devant notre première instruction assembleur. Elle utilise une abréviation du mot anglais PUSH (pousser). La lettre U qui termine PSHU correspond au registre U (ce pourrait être tout aussi bien S si l'exemple mettait en œuvre l'autre pile), et la lettre A précise que c'est le contenu de l'accumulateur A qui va être placé sur la pile. Lorsque cette instruction aura été exécutée par le microprocesseur, voici ce qui se sera passé :

Le registre A n'aura subi aucune modification. Son contenu, le nombre 20, sera allé s'écrire dans la pile mais ce registre en aura conservé la trace. Il en sera toujours ainsi quand nous donnerons l'ordre au processeur de transférer un nombre d'un registre vers une case mémoire : le registre ne sera pas modifié, et tout se passera comme si le registre n'avait envoyé qu'un double, un duplicata à la mémoire.

Les registres B et X ne sont pas intervenus dans l'instruction et gardent donc la même valeur.

Le registre U est en fin de compte le seul qui sera modifié. Il va passer à 32699 indiquant de ce fait que la pile se trouve maintenant à cette adresse. Voici donc défini le rôle de U : il contient un nombre de 16 bits et ce nombre est l'adresse de l'octet où se trouve la pile.

Pour l'instant, notre pile n'est formée que d'un seul octet (à l'adresse 32699) et ne contient qu'un seul nombre (20).

	PILE	REGISTRE U
Octet 32699	20 (décimal)	32699

Continuons avec **PSHU B**

Cette fois, c'est le contenu du registre B qui va aller se placer sur la pile — c'est-à-dire dans l'octet 32698. Si nous avions la possibilité de mélanger à loisir l'assembleur et le BASIC, nous taperions au clavier PRINT PEEK (32698) : la réponse serait 35.

Naturellement, le registre U est encore modifié, n'oublions pas que c'est lui qui tient les comptes de la pile.

	PILE	REGISTRE U
Octet 32698	35 (décimal)	32698
Octet 32699	20 (décimal)	

Et si on essayait **PSHU X** ?

Il va falloir que le microprocesseur aille ranger dans la pile la valeur de X. Or X est un registre 16 bits et il ne peut être question d'en placer le contenu sur un seul octet. Par contre, en nous servant des deux octets 32696 et 32697, nous aurons la réponse à notre problème : les huit bits de gauche de X (bits de poids fort) seront recopiés dans l'octet 32696 et les huit derniers bits (bits de poids faible) seront inscrits dans l'octet 32697.

	PILE	REGISTRE U
Octet 32696	15 (décimal)	
Octet 32697	129 (décimal)	
Octet 32698	35 (décimal)	
Octet 32699	20 (décimal)	32696

15 est la traduction décimale de la partie gauche de X et 129 est la traduction, toujours décimale, de sa partie droite.

Avant de passer à l'opération de dépilement, notons bien que les nombres sont stockés dans la pile sur des octets dont les adresses décroissent à chaque fois. Il n'y a rien à y faire, c'est dans la logique sans doute de notre ordinateur : lui, il empile par en dessous. Le tout est de le savoir.

PULU X

C'est notre deuxième instruction assembleur : elle effectue exactement le travail inverse de la première. Elle va rechercher un nombre dans la pile et elle l'écrit dans le registre X. Elle a donc dépilé les octets 32696 et 32697 et a été les replacer dans X. Par là même, notre pile ne contient plus que deux octets et le registre U repasse à 32698.

	PILE	REGISTRE U
Octet 32698	35 (décimal)	
Octet 32699	20 (décimal)	32698

L'intérêt des instructions PSHU et PULU n'apparaît pas immédiatement aux programmeurs qui débutent avec l'assembleur car ils ne voient pas à quoi peuvent bien servir deux choses qui ne font rien d'autre que nous ramener à notre point de départ. Et pourtant c'est là justement qu'en réside tout l'intérêt : la principale difficulté de l'assembleur vient de ce que nous ne disposons que d'un nombre très réduit de registres. Nous verrons très vite le très gros avantage qu'il y a à placer la valeur d'un registre dans la pile (PUSH), à utiliser ce registre pour faire autre chose, et à retrouver (PULL) la valeur initiale de ce même registre.

Finissons-en avec notre exemple :

PULU A

La machine va aller écrire 35 dans l'accumulateur. Elle exécute très exactement l'ordre qu'on lui donne en prenant le nombre écrit au sommet de la pile — donc 35 — et en le plaçant dans A. A ce moment-là les deux registres A et B contiennent précisément la même valeur, 35.

	PILE	REGISTRE U
Octet 32699	20 (décimal)	32699

PULU B

La pile que nous avions constituée est réduite à zéro. Le registre B va prendre la valeur 20 et le registre U reprendra sa valeur initiale 32700.

En fin de compte, dans notre exercice, les accumulateurs auront été échangés et les registres X et U auront retrouvé les contenus qu'ils avaient au départ.

Un autre exemple maintenant. On repart du même énoncé :

A contient le nombre décimal 20
B contient le nombre décimal 35
X contient le nombre décimal 3969
U contient le nombre décimal 32700.

Puis, on exécute les instructions suivantes :

```
PSHU A  –  PSHU B  –  PSHU X
PULU A  –  PULU B  –  PULU X
```

Nous laisserons au lecteur le soin de déterminer ce que les registres A, B et X contiendront à la fin de ces six opérations.

(Réponses : A (15) ; B (129) ; X (8980))

On peut très bien se servir de l'autre pile, S, dans les programmes assembleurs. Elle se met en œuvre exactement de la même façon que U, mais il faudra se souvenir qu'elle est utilisée aussi par le microprocesseur. Les ennuis nous seront garantis si, par mégarde, nous ne remettons pas la pile système dans l'état où nous l'avons trouvée. Si on a eu besoin par exemple d'empiler trois octets dans S, il faudra être sûr qu'à la fin de notre programme les trois octets en question ont été dépilés.

4. Le registre PC

C'est un registre 16 bits que l'on appelle le compteur de programme ou le compteur ordinal. Le nombre qui est écrit dans ce registre est l'adresse de la prochaine instruction à exécuter. Il permet au microprocesseur de toujours savoir à quel octet du programme il va devoir s'intéresser. On ne l'utilise en programmation que dans des cas bien particuliers : il n'est pas directement accessible.

LES MODES D'ADRESSAGE

Un mode d'adressage est un moyen qui permet au microprocesseur d'avoir accès à une donnée. Cette donnée peut être un nombre quelconque dont on aura besoin dans le programme, un nombre qui se trouve déjà dans un registre, ou encore un nombre qui se trouve écrit quelque part en mémoire.

La connaissance des principaux modes d'adressage est obligatoire : elle permet d'écrire les programmes de la façon la plus courte, la plus simple et la plus lisible possible.

1. L'adressage inhérent

L'adressage inhérent est habituellement réservé aux instructions qui agissent directement sur les valeurs contenues par les registres. Ces instructions se comprennent d'elles-mêmes et n'ont aucunement besoin qu'on leur ajoute des indications.

Exemple : INCA

Tous les microprocesseurs comprennent ce genre d'instruction : elle signifie que le registre A se verra incrémenté, c'est-à-dire que la valeur qu'il contenait se retrouvera augmentée d'une unité.

A contenait 35 (par exemple)

```
INCA
```

A contient 36.

2. L'adressage immédiat

Dans ce mode, une valeur apparaît après l'instruction assembleur. Prenons par exemple : LDA # 5

La formule LDA, qui sera retrouvée tout au long de ce livre, signifie que l'on va placer (charger) un nombre dans le registre A. Il est facile de voir qu'ici l'instruction LDA n'aurait pas pu être écrite toute seule, comme dans l'adressage inhérent. Il nous faut absolument rajouter des indications à la suite : et, si l'on doit mettre un nombre dans l'accumulateur A, il faut bien dire lequel.

Dans le mode d'adressage immédiat, c'est la valeur marquée après l'instruction (ici 5) qui sera écrite dans A.

A contenait par exemple 50

```
LDA # 5
```

A contient alors 5.

Le signe # est réservé à l'adressage immédiat et permet de ne pas avoir, après chaque instruction, à écrire sous quel mode elle doit être comprise.

Un contre-exemple :

```
LDA # 300
```

Cette ligne devrait, en principe, écrire dans A le nombre 300. Vous l'aviez deviné ; ceci n'a aucun sens puisque A est un registre huit bits et que le nombre maximum qu'il peut contenir est 255.

3. L'adressage étendu

On l'appelle souvent adressage absolu car c'est un mode qui va concerner le contenu de n'importe quel octet de la mémoire.

```
LDA >50
```

L'accumulateur sera chargé non par le nombre 50 comme il l'aurait été dans l'adressage immédiat, mais avec la valeur écrite dans l'octet numéro 50.

```
PRINT PEEK (50) ; réponse : 8
```

En fin de compte, le registre A contiendra 8.

C'est le signe > qui sera toujours utilisé quand l'instruction sera écrite avec le mode étendu.

Un deuxième exemple :

```
LDA >60000
```

Un piège, comme dans le paragraphe précédent ? 60000 paraît bien

trop important pour notre accumulateur huit bits. Mais non, cette ligne n'est pas un non-sens : elle signifie que le registre A va contenir non pas le nombre 60000, mais le nombre qui se trouve écrit dans l'octet ayant 60000 pour adresse.

PRINT PEEK (60000) ; réponse : 126

Ce qui fait que A sera chargé, une fois l'instruction assembleur exécutée, par la valeur 126.

4. L'adressage indexé

C'est un mode qui concerne les contenus des registres X, Y, U et S. Un peu délicat à utiliser au départ, il s'avère ensuite offrir de multiples possibilités au programmeur.

Voici quelques exemples :

LDA , X

L'accumulateur va être chargé avec la valeur qui se trouve dans l'octet ayant pour adresse le nombre écrit dans X.

Supposons que dans le registre 16 bits X il y ait le nombre 50000.

PRINT PEEK (50000) ; réponse : 159.

Après LDA , X il y aura dans le registre A la valeur 159. Cette instruction est donc tout à fait équivalente à LDA > 50000 (adressage étendu).

Deuxième exemple :

LDA 3, X

Cette fois, le processeur va aller chercher, pour l'écrire dans A, la valeur qui se trouve en mémoire dans l'octet ayant pour adresse le nombre contenu par X auquel on ajoute 3.

Admettons ici aussi que X contienne la valeur 50000.

On ajoute 3 à 50000 et l'octet concerné est alors le numéro 50003.

PRINT PEEK (50003) ; réponse : 112.

Ainsi, dans cet exemple, A va être chargé avec le nombre 35.

Faisons un nouvel essai :

```
LDA B, X
```

On suppose que X contient toujours la valeur 5000 et que l'accumulateur B, pour sa part, contient le nombre 80. Alors A sera chargé avec le nombre écrit dans l'octet 5080.

PRINT PEEK (5080) ; réponse : 58.

A contiendra donc la valeur 58.

Voyons un exemple de plus :

```
LDA ,X+
```

En considérant toujours que X contient le nombre 5000, cette instruction indique au processeur qu'il doit placer dans A le nombre écrit dans l'octet 5000. Nous savons déjà que le PEEK de cet octet vaut 4. A contiendra donc la valeur 4. Quant au signe + écrit à droite de X, il signifie qu'une fois l'instruction exécutée le registre X devra être incrémenté : il passera alors à 5001. Si deux signes + suivaient X, ce registre verrait sa valeur augmentée de deux.

Un dernier exemple :

```
LDA , -X
```

Vous vous en doutez, cette fois le registre X va être décrémenté. Mais attention, la décrémentation intervient avant que LDA ne soit exécutée. Si X vaut encore 5000, alors l'accumulateur contiendra la valeur de l'octet 4999.

PRINT PEEK (4999) ; réponse : 37.

C'est 37 qui sera écrit dans A.

Puisque nous en sommes à l'adressage indexé, prenons la peine de comprendre comment les codes machine correspondants pourront être obtenus. Dans un premier temps, le lecteur pourra directement pas-

ser au chapitre suivant : ce n'est que plus tard, lorsque le besoin s'en fera sentir, qu'il pourra revenir à ce passage délicat.

Comment coder, par exemple, **LDA , X**

On cherche le code de LDA indexé dans l'Appendice A : A6.

Le tableau de l'Appendice C nous fournit le code suivant :

Forme assembleur	Post-octet
,R	1RR00100

Puisque R est pour nous le registre X, on aura : RR = 00 (ceci se lit au-dessous du tableau).

Le post-octet vaut donc : 10000100 ou 84 hexa.

Ainsi LDA ,X se code A6 84.

Recommençons avec **LDA , Y**

Cette fois, R est le registre Y et on voit, au-dessous du tableau, que RR = 01.

D'où la valeur du post-octet : 10100100 soit A4 hexa.

Notre instruction sera codée : A6 A4.

Cherchons maintenant à coder **LDA 5 , X :**

Forme assembleur	Post-octet	
n , R	0RRnnnnn	Sur 5 bits
n , R	1RR01000	Sur 8 bits

On peut faire ce codage sur 5 bits ou sur 8 bits.

— sur 5 bits : n = 5 , R = X et RR = 00

Le post-octet s'écrira 00000101, les cinq derniers bits servent à écrire en binaire *n*, dans le mode complément à deux s'il est négatif, sur 5 bits : n = 5 = 00101

L'instruction sera donc codée A6 05 en hexadécimal.

— sur 8 bits : n = 5 , R = X et RR = 00

Le post-octet va s'écrire alors 10001000, c'est-à-dire 88 hexa. Il faut noter que dans le post-octet, cette fois, la valeur *n* n'apparaît pas. Comme il est nécessaire de la faire connaître au microprocesseur, un octet supplémentaire contenant justement cette valeur en complément à deux sera obligatoire.

Voici le codage : A6 88 05.
Exemple suivant : codage de **LDA — 50 , U**

Forme assembleur	Post-octet	
n , R	1RR01000	Sur 8 bits

En tenant compte du fait que n = −50, R = U et RR = 10, on obtient comme post-octet :

11001000 ou C8 hexa.

Codons maintenant −50 en complément à deux sur 8 bits :

00110010 ←——— 50 décimal
11001101 ←——— inversion de tous les bits
11001110 ←——— addition de 1

Ainsi −50 s'écrit CE en hexadécimal.
Nous obtenons alors le codage de LDA −50 , U :

```
A6 C8 CE
```

Un dernier exemple : soit à coder **LDA , --S**

Forme assembleur	Post-octet
, --S	1RR00011

Ici seul le registre S est en cause, aucun nombre n'intervient. L'instruction se notera au total sur deux octets.
Puisqu'ici R = S et RR = 11, on obtient comme post-octet :

11100011 ou E3 en hexadécimal.

Les codes machine de l'instruction étudiée sont donc A6 E3.

5. L'adressage indexé indirect

Il se note comme l'adressage indexé normal, mais en plus, des crochets sont placés de chaque côté de ce qui correspond au post-octet :

```
LDA [, X]
```

Pour comprendre cet exemple, supposons, une fois encore, que le registre X contienne la valeur 5000. L'accumulateur ne sera pas chargé dans ce cas par le nombre contenu dans l'octet 5000, mais par le nombre contenu dans l'octet dont l'adresse sera calculée à partir des octets 5000 et 5001. Voici comment :

PRINT PEEK (5000) ; réponse : 4 soit 00000100 sur 8 bits
PRINT PEEK (5001) ; réponse : 129 soit 10000001 sur 8 bits

En associant les octets 5000 et 5001, on obtient le nombre de 16 bits :

0000010010000001

La traduction de cette valeur en décimal donne 1153 :

PRINT PEEK (1153) ; réponse : 68

Ce qui fait donc que le registre A va contenir au bout du compte le nombre 68.

On peut résumer en disant que l'accumulateur a été chargé d'une façon indirecte, en passant par une adresse obtenue à partir du registre X.

Nous en avons terminé avec ce chapitre, le cap difficile de la théorie est passé. Asseyons-nous devant notre TO 7 et voyons comment nous allons l'obliger à comprendre puis à exécuter un programme écrit en langage machine.

4
ÉTUDE D'UN EXEMPLE

Rentrez dans votre TO 7 le programme suivant et faites-le exécuter :

```
10 CLEAR, 32700 : FOR I = 32701 TO 32706 : READ I$ :
   POKE I, VAL ( "&H" + I$ ) : NEXT
20 DATA 86 , FF , B7 , 4E , 20, 39
30 EXEC 32701
```

Vous voyez apparaître, vers le milieu de votre écran, un petit segment bleu. Ce segment n'a pu être allumé par POKE, comme dans le Chapitre 2, nous le retrouverions au milieu du programme BASIC. Nous l'avons obtenu grâce à l'exécution de notre premier programme écrit en langage machine.

Lignes	Codes machine	Assembleur		
1	86 FF	LDA	# 255	(immédiat)
2	B7 4E 20	STA	> 20000	(étendu)
3	39	RTS		(inhérent)

C'est cette présentation que nous reprendrons tout au long de ce livre. Pour l'heure, faisons l'analyse minutieuse de tout ce qui est nouveau.

PARTIE ASSEMBLEUR

C'est celle que le lecteur assimilera le plus vite.

```
LDA # 255
```

Nous retrouvons une instruction qui a déjà été rencontrée. Elle signifie que le registre A va contenir la valeur décimale 255. Le symbole # est là pour indiquer le mode d'adressage immédiat. On ignore quel nombre se trouvait dans l'accumulateur avant cette instruction, mais maintenant on est sûr de la valeur de A : c'est 255 en décimal, ou FF en hexadécimal, ou encore 11111111 en binaire.

```
STA >20000
```

STA est une instruction très fréquemment utilisée en assembleur : elle signifie que la valeur contenue dans A va devoir être inscrite dans un octet de la mémoire. STA est l'abréviation de STORE A qui, en anglais, veut dire RANGER A. Le mode d'adressage choisi, l'étendu, repéré par le signe >, nous laisse entendre que 20000 est l'adresse d'un octet de la mémoire. En définitive, c'est dans cet octet que sera rangé le contenu de A. Cette instruction est donc l'équivalent assembleur de la ligne BASIC POKE 20000, 255 car, ne le perdons pas de vue, le registre A contient le nombre 255.

Voici donc expliqué pourquoi un segment s'est allumé sur notre écran quand le programme a été exécuté. Ce segment correspond à l'octet numéro 20000, octet qui se trouve dans la zone mémoire écran. Comme 255 s'écrit en binaire sous la forme d'un nombre constitué de huit chiffres 1, on comprend que tous les points de ce segment aient été allumés.

```
RTS
```

Cette instruction, souvent écrite en dernière ligne, sert à indiquer au microprocesseur que le programme assembleur est terminé. C'est le BASIC qui reprend alors le contrôle de la situation.

CODES MACHINE

Un ordinateur ne comprend que des nombres et pour lui les expressions LDA, STA ou RTS ne veulent absolument rien dire. Il va donc falloir lui traduire le programme que nous avons écrit en assembleur sous la seule forme qui lui soit compréhensible : les codes machine.

Insistons bien sur la différence qu'il y a entre les langages assembleur et machine : le premier est conçu pour l'esprit humain et il est formé d'instructions qui ont un sens pour nous. Quand on écrit LDA, on sait très bien ce qui se passera dans le processeur, on n'a pas besoin de faire un gros effort pour comprendre que le contenu du registre A sera modifié et remplacé par une nouvelle valeur. La forme assembleur permet d'écrire des programmes qui soient lisibles, des programmes qui soient écrits avec des mots ou des abréviations dont on s'habituera très vite à connaître le sens. Qui, parmi nous, serait capable d'interpréter la suite des codes machine 86, FF, B7, 4E... qui, de surcroît, sont écrits en hexadécimal ?

Livrons-nous donc à la traduction en langage machine du programme. C'est un exercice plutôt simple, mais attention à la moindre erreur de codage qu'il sera ensuite très difficile de retrouver ! Servez-vous du tableau de l'Appendice A.

86 est l'équivalent pour la machine de LDA. Vous constatez que LDA se code de différentes façons suivant le mode d'adressage. Celui qui nous intéresse est l'immédiat, dans la première colonne donc. Il faudra toujours se souvenir que les codes machine sont écrits en hexadécimal ; 86, ainsi que tous les autres codes de ce tableau, respecte cette règle.

FF est le nombre obtenu en convertissant 255 en hexadécimal. Le microprocesseur, après avoir interprété 86, s'attendra à ce qu'on lui dise avec quel nombre il doit charger A. Puisque FF vient à la suite de 86, il comprendra que la valeur FF (ou 255 décimal) doit être placée dans l'accumulateur.

B7 est le code machine de STA. Il doit être choisi dans la bonne colonne, celle de l'adressage étendu. C'est en effet ce mode que nous avons décidé d'utiliser en écrivant le programme assembleur. Quand l'ordinateur va lire ce code, il saura qu'il lui faut alors s'intéresser aux deux valeurs suivantes.

4E 20 ne forment en réalité qu'un seul nombre hexadécimal : 4E20 (soit 20000 en décimal). Le lecteur comprendra avant la fin de ce chapitre pourquoi nous avons écrit séparément 4E et 20. La seule chose à assimiler pour l'instant est la suivante : la machine, ayant rencontré B7, comprendra qu'elle doit aller placer le contenu du registre A dans un octet de la mémoire. Or, l'adresse des octets ne peut s'obtenir avec seulement deux chiffres hexadécimaux ; quatre sont en effet nécessaires, quatre mais quatre seulement puisque le dernier octet de la mémoire a pour numéro FFFF.

39 correspond à l'instruction RTS. Il n'y a pas de risque à choisir dans le tableau la mauvaise colonne, le seul mode possible d'adressage étant l'inhérent. Ce nombre termine la série des codes machine.

PROGRAMME BASIC

CLEAR est une instruction qui permet de réserver de la place en mémoire. Elle est utilisée pour pouvoir fournir au programme une place

suffisante dans les manipulations de chaînes, pour définir le nombre de caractères graphiques GR$ et aussi pour indiquer à l'ordinateur qu'il ne devra pas se servir d'un certain nombre d'octets de la mémoire.

```
CLEAR, 32700
```

doit être interprété de la façon suivante : l'ordinateur pourra utiliser n'importe quel octet dont l'adresse est inférieure ou égale à 32700 mais ne devra en aucun cas modifier les contenus des octets 32701, 32702, 32703, ... 32767. Ce dernier nombre est l'adresse de la fin de la mémoire vive du TO 7 non muni de l'extension 16 K.

```
FOR I = 32701 TO 32706
```

Nous avons besoin de ces six octets et nous sommes certains, grâce à CLEAR, qu'ils seront réservés à notre usage.

```
READ I$
```

La lecture des six valeurs écrites en DATA sera effectuée : il est nécessaire de faire lire une chaîne de caractères car, d'une part, des lettres apparaissent dans la ligne 30 et, d'autre part, les chiffres eux-mêmes sont écrits en hexadécimal et non en décimal.

```
POKE I, VAL ( "&H" + I$ )
```

Au début de la boucle, I vaut 32701 et I$ vaut «86». L'interpréteur reconnaît donc le nombre hexadécimal &H86 et il en écrit la valeur dans l'octet 32701. Au deuxième passage, le nombre FF sera placé dans l'octet 32702. Après le dernier passage, les six codes machine seront inscrits dans les octets allant de 32701 à 32706. On comprend ainsi pourquoi le nombre 4E20 (20000 en décimal) a été écrit en deux parties que l'on a placées, l'une dans l'octet 32704, l'autre dans l'octet 32705.

```
NEXT
```

Voilà notre programme chargé en mémoire centrale et il ne reste plus qu'à l'exécuter.

```
EXEC 32701
```

D'une manière identique à RUN qui lance un programme BASIC, EXEC démarre l'exécution du programme machine. La commande des opérations passe au microprocesseur qui va réaliser les instructions correspondant aux valeurs qui se trouvent dans les octets 32701 et suivants. En rencontrant 86, il va charger l'accumulateur A avec le nombre qui se trouve dans l'octet d'après et continuer ainsi jusqu'à la valeur 39.

REMARQUE

On peut se demander quel est le rôle des octets dont les adresses vont de 32707 à 32767. Il est nul, ces octets ne sont utilisés ni par l'ordinateur puisque CLEAR l'en empêche, ni par nous car, en fin de compte, notre programme n'a besoin que de six octets. La logique aurait voulu que l'on écrive le BASIC avec les modifications suivantes :

```
10 CLEAR, 32761 : FOR I = 32762 TO 32767 ...
30 EXEC 32762
```

La ligne 10 aurait placé dans les octets 32762 à 32767 les six codes machine que le processeur aurait retrouvé avec EXEC 32762. Nous avons choisi de bloquer tous les octets à partir de 32701 pour une raison bien simple : la grande majorité des programmes de ce livre sera logée en mémoire à partir de cette adresse afin d'avoir une présentation uniforme. Et ce premier exemple respecte cette règle. Ajoutons que la perte d'une soixantaine d'octets ne doit pas être considérée comme excessive et qu'il est tout de même plus facile, si l'on a un programme de 19 octets par exemple, d'écrire

```
FOR I = 32701 TO 32719
```

plutôt que

```
FOR I = 32749 TO 32767.
```

5

ÉLÉMENTS DE PROGRAMMATION DU 6809

Note importante

Le lecteur ne devra pas oublier, dans chacun des programmes qui sont donnés à titre d'exemples, d'inclure les lignes 10 et 20 suivantes :

```
10 CLEAR, 32700 : FOR I = 32701 TO 327?? : READ I$ :
   POKE I, VAL ( "&H" + I$ ) : NEXT
20 DATA ....
```

Les deux points d'interrogation de la ligne 10 devront être remplacés par le nombre d'octets du programme en langage machine et la ligne 20 devra contenir en DATA la liste des codes machine écrits sous forme hexadécimale.

LDA

Abréviation de LOAD A (charger A), cette instruction permet de placer une valeur 8 bits dans le registre A.

Les modes d'adressage possibles sont l'immédiat, l'étendu et l'indexé.

Exemple : MODE D'ADRESSAGE IMMÉDIAT (6 octets)

1. Programme BASIC

```
10 CLEAR, 32701 : FOR I = 32701 TO 32706 : READ I$ :
   POKE I, VAL ( "&H" + I$ ) : NEXT
20 DATA 86, 00, B7, E7, C1, 39
30 EXEC 32701
```

Les lignes 10 et 20 sont données dans cet exemple mais seuls leurs numéros apparaîtront dans le reste de ce livre. Pensez à chaque fois à les compléter.

2. Programme assembleur

Lignes	Codes machine	Assembleur		
1	86 00	LDA	# 0	Immédiat
2	B7 E7 C1	STA	> 59329	Etendu
3	39	RTS		Inhérent

Ligne 1 : la valeur 0 est mise dans l'accumulateur A. 86 est le code hexadécimal de l'instruction LDA en mode immédiat.

Ligne 2 : B7 (hexa) est le code de l'opération STA en mode étendu et E7C1 n'est rien d'autre que l'écriture hexadécimale du nombre 59329. Cette ligne va permettre de ranger dans l'octet 59329 la valeur contenue par le registre A. Il y aura donc 0 dans l'octet en question.

Ligne 3 : 39 est le code de l'instruction RTS qui indique que le programme machine est terminé. On retourne alors au BASIC.

La présentation du programme sous forme de tableau n'est faite que dans un souci de clarté. Contrairement au BASIC, les numéros de ligne n'ont aucune importance pour le microprocesseur qui ne s'intéresse, dans cet exemple, qu'à la suite des octets 32701 à 32706.

Après l'exécution de ce programme, vous verrez votre ordinateur rester sans voix. Son bip désormais familier aura disparu, l'appui sur une touche ne donnera plus le son habituel. Ceci provient du fait que nous avons, en langage machine, écrit 0 dans l'octet 59329 et cette valeur rend aphone le TO 7. Pour le réentendre, un seul moyen : écrire 56 (décimal) dans l'octet 59329. Naturellement, ceci peut se faire avec l'instruction POKE mais essayez plutôt de modifier le programme assembleur pour obtenir l'effet recherché : vous aurez alors conçu votre premier programme en langage machine.

LDB

Cette instruction est absolument identique à LDA, mais elle concerne le chargement du registre B. Là encore, les modes d'adressage possibles sont l'immédiat, l'étendu et l'indexé.

LDX

Abréviation de LOAD X (charger X), cette instruction permet de placer une valeur 16 bits dans le registre X. Les modes d'adressage possibles sont l'immédiat, l'étendu et l'indexé.

Exemple : MODE D'ADRESSAGE IMMÉDIAT (10 octets)

1. Programme BASIC

```
 5 CLS
10
20
30 EXEC 32701
40 LOCATE 0, 10 : PRINT "C'EST FINI"
```

2. Programme assembleur

Lignes	Codes machine	Assembleur		
1	8E 7F C6	LDX	# 32710	Immédiat
2	A6 84	LDA	, X	Indexé
3	B7 60 39	STA	> 24633	Etendu
4	39	RTS		Inhérent
5	06 (donnée)			

Ligne 1 : le registre X va contenir la valeur 32710. N'oubliez pas que X est un registre 16 bits et que, contrairement au registre A qui ne peut contenir un nombre supérieur à 255, il est possible de placer dans X n'importe quel nombre inférieur ou égal à 65535 (16 fois le chiffre 1 en binaire, c'est-à-dire $2^{16} - 1$).

Ligne 2 : A sera chargé par la valeur qui se trouve dans l'octet dont le numéro est indiqué par X, c'est-à-dire par la valeur qui se trouve dans l'octet 32710. Par POKE (ligne 10), le nombre 6 a été écrit dans cet octet.

Ainsi donc, le registre A va contenir 6, ce qui fait que l'on aurait pu remplacer cette ligne 2 par LDA #6 ou par LDA >32710.

Ligne 3 : la valeur 6 est rangée dans l'octet 24633. Comme, à chaque fois que l'on écrit le nombre 6 dans cet octet, l'ordinateur double la grandeur des caractères, nous avons retrouvé ici la fonction BASIC ATTRB 1, 1. La ligne BASIC 40 en donne la confirmation puisque le message «C'EST FINI» s'inscrit au milieu de l'écran en lettres de grandeur double.

Ligne 4 : l'instruction RTS programme le retour au BASIC et le microprocesseur ne cherche donc pas à interpréter la valeur écrite à la ligne 5.

LDY LDD LDU LDS

Ces quatre instructions s'utilisent de la même façon que LDX. Elles servent à placer dans les registres Y, D, U et S des valeurs de 16 bits.

STA

Abréviation de STORE A (ranger A), STA permet de placer dans un octet de la mémoire la valeur 8 bits qui a précédemment été chargée dans le registre A. Deux modes d'adressage : les adressages indexé et étendu.

Exemple : MODE D'ADRESSAGE INDEXÉ (16 octets)

1. Programme BASIC

```
10                    I
20
30 EXEC 32701
40 PRINT "L'OCTET 32760 CONTIENT LA VALEUR";PEEK(32760)
50 PRINT "L'OCTET 32762 CONTIENT LA VALEUR";PEEK(32762)
```

2. Programme assembleur

Lignes	Codes machine	Assembleur		
1	C6 0A	LDB	#10	Immédiat
2	86 02	LDA	#2	Immédiat
3	8E 7F EE	LDX	#32750	Immédiat
4	A7 85	STA	B,X	Indexé
5	10 8E 7D F0	LDY	#32240	Immédiat
6	E7 AB	STB	D,Y	Indexé
7	39	RTS		Inhérent

L'instruction STA ayant déjà été rencontrée et son action analysée, intéressons-nous plutôt dans cet exemple aux facilités de programmation qu'offre l'adressage indexé.

Lignes 1 et 2 : les accumulateurs B et A sont respectivement chargés par les valeurs 10 et 2 et, par conséquent, le nombre qui se trouve dans le registre D est alors connu : c'est 522.

Registre A (2 décimal)								Registre B (10 décimal)							
0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	0

Registre D (522 décimal)

Ligne 4 : on range la valeur de A(2) dans l'octet dont l'adresse se calcule en ajoutant les nombres contenus dans X(32750) et dans B(10). La ligne BASIC 40 établira cet état de choses : l'octet 32760 contient bien la valeur 2. A noter que cette valeur reste inscrite dans l'accumulateur A, l'instruction STA ne l'ayant pas effacée.

Ligne 5 : on place dans le registre Y le nombre 32240. Il faut noter que les codes machine sont ici au nombre de quatre ; les deux derniers 7D et F0 sont la traduction hexadécimale de 32240 et les deux premiers 10 et 8E correspondent au seul ordre LDY en mode immédiat. On rencontrera dans ce livre quelques autres instructions qui doivent s'écrire sur 2 octets.

Ligne 6 : on stocke le nombre 10. Il s'agit cette fois de STB — dans l'octet ayant pour adresse 32240 (registre Y) + 522 (registre D) soit 32762. Nous en aurons la confirmation avec la ligne 50 du programme BASIC :

```
PEEK(32762) = 10
```

Cet exemple n'a qu'un intérêt : apprendre au lecteur à intervenir sur n'importe quel octet de la mémoire. Il aurait été aisé d'écrire le programme d'une façon plus courte mais nous avons tenu à utiliser un mode d'adressage indexé qui ajoute à X et à Y le contenu d'un accumulateur. Bien entendu, les nombres 32240, 32750, 32760 et 32762 ont été choisis pour la commodité des explications mais d'autres valeurs auraient très bien pu être prises.

STB

Cette instruction, identique dans son principe d'utilisation à STA, commande le passage d'une donnée 8 bits du registre B vers un octet de la mémoire.

STX

La valeur 16 bits contenue dans le registre X est rangée en mémoire. Le rangement s'effectue sur deux octets consécutifs. On peut utiliser les modes d'adressage étendu et indexé.

Exemple : MODE D'ADRESSAGE ÉTENDU ET INDEXÉ (11 octets)

1. Programme BASIC

```
10
20
30 EXEC 32703
40 PRINT "L'OCTET 32750 CONTIENT LE NOMBRE";PEEK(32750)
50 PRINT "L'OCTET 32751 CONTIENT LE NOMBRE";PEEK(32751)
60 PRINT "LE REGISTRE X CONTENAIT DONC LE NOMBRE
   ";256*PEEK(32750) + PEEK(32751)
```

2. Programme assembleur

Lignes	Codes machine	Assembleur		
1	4E 20 (donnée)			
2	BE 7F BD	LDX	>32701	Etendu
3	BF 7F EE	STX	>32750	Etendu
4	AF 84	STX	,X	Indexé
5	39	RTS		Inhérent

Ligne 1 : dans cet exemple, ce sont les données qui ont été écrites en premier, 78 (4E Hexa) est placée dans l'octet 32701 et 32(20 Hexa) dans l'octet 32702. Naturellement, puisque le programme en langage machine ne commence qu'à la ligne 2, nous avons pris soin de taper la ligne 30 en conséquence : EXEC 32703.

Ligne 2 : X est chargé avec la valeur contenue dans l'octet 32701. X étant un registre 16 bits, il n'est pas question de le remplir avec un seul octet et il faudra donc se servir d'un deuxième octet : le numéro 32702.

octet 32701 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 0 |

soit 4E en hexa et 78 en décimal

octet 32702 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |

soit 20 en hexa et 32 en décimal

Après exécution de la ligne 2,X contiendra :

0	1	0	0	1	1	1	0	0	0	1	0	0	0	0	0

soit 4E20 en hexa et 20000 en décimal

On a l'habitude de considérer les 8 bits de gauche comme l'octet de poids fort du registre X et les 8 bits de droite comme l'octet de poids faible. Ceci permet d'obtenir la valeur décimale contenue dans ce registre à l'aide de la relation suivante : 256*octet de poids fort (en décimal) + octet de poids faible (en décimal). Pour ce qui concerne notre exemple, c'est le nombre 20000(256*78 + 32) qui est maintenant placé dans X.

Ligne 3 : le contenu du registre est rangé à l'adresse 32750. Pour les mêmes raisons que précédemment, il sera nécessaire d'utiliser un deuxième octet : la partie forte de X (78 en décimal) se retrouve donc l'octet 32750 et sa partie faible (32 en décimal) dans l'octet 32751. Les lignes 40, 50 et 60 du programme BASIC nous permettent de retrouver tout ceci :

```
PEEK(32750) =78
PEEK(32751) =32
REGISTRE X=20000
```

Ligne 4 : X contenant toujours la même valeur, il s'agit maintenant de la stocker dans l'emplacement mémoire dont l'adresse est donnée par X. En clair, il faut donc ranger la valeur 20000 dans les 2 octets nécessaires pour cela, octets qui ont pour adresses 20000 et 20001. La partie forte de X sera écrite dans le premier et la partie faible dans le second. Les deux octets étant placés dans la zone mémoire écran, on verra les deux segments correspondants s'allumer sur l'écran du téléviseur.

STY STD STU STS

On utilise ces quatre instructions de la même façon que STX. Elles permettent toutes de placer le contenu du registre spécifié dans un emplacement constitué par 2 octets de la mémoire.

ADDA

Cette instruction va ajouter les contenus du registre A et d'un octet mémoire. Le résultat de l'addition sera alors placé dans A. On peut utiliser les modes d'adressage immédiat, indexé et étendu.

Exemple : MODE D'ADRESSAGE INDEXÉ (11 octets)

1. Programme BASIC

```
10
20
30 INPUT "PREMIER NOMBRE";N:POKE 32750,N
40 INPUT "DEUXIEME NOMBRE";M:POKE 32751,M
50 EXEC 32701
60 PRINT "LA SOMME VAUT";PEEK(32752) :GOTO 30
```

2. Programme assembleur

Lignes	Codes machine	Assembleur	
1	8E 7F EE	LDX	#32750
2	A6 84	LDA	,X
3	AB 88 01	ADDA	1,X
4	A7 02	STA	2,X
5	39	RTS	

Ligne 2 : le registre A est chargé avec la valeur contenue dans l'octet pointé par X, c'est-à-dire l'octet 32750. La ligne BASIC 30 a écrit dans cet octet le premier terme de la somme. N ne doit pas bien sûr dépasser 255 sinon le message FC ERROR apparaîtrait sur votre écran.

Ligne 3 : ADDA 1,X signifie que l'on ajoute la valeur de A et la valeur inscrite dans l'octet ayant pour adresse X + 1. De ce fait cette ligne additionne le premier et le second nombre de la somme, second nombre qui a été placé par POKE dans l'octet 32751 (ligne BASIC 40).

Ligne 4 : le résultat de l'opération étant dans le registre A il ne reste plus qu'à ranger la somme obtenue dans l'octet 32752 puisque c'est là que le BASIC ira chercher la réponse.

Faisons tourner le programme

Premier nombre :	Deuxième nombre :	Somme :
10	20	30
100	0	100
200	60	4
250	250	244

Il n'y a rien à redire pour les deux premiers cas : les résultats obtenus sont conformes à nos prévisions. Quant aux calculs suivants, il ne sera pas bien compliqué d'établir leur cohérence : puisque A est un registre 8 bits, le nombre le plus grand que l'on puisse écrire est 11111111 (255 en décimal). Si, à ce moment-là on essaie d'ajouter 1, le registre repassera à 00000000 (0 en décimal) et c'est ce qui explique que 260 soit devenu 4 dans notre troisième somme. D'une manière analogue, en ajoutant 250 et 250 on ne trouve pas 500 mais 500-256 soit 244.

Avant de passer à la suite de votre étude, essayez de retrouver, à l'aide de l'annexe donnée à la fin de ce livre, comment les codes machines des lignes 3 et 4 ont été obtenus.

Assurez-vous d'avoir bien compris que l'on aurait tout aussi bien pu

— coder ADDA 1,X sur 2 octets : AB 01

— coder STA 2,X sur 3 octets : A7 88 02

ADDB

On procède avec cette instruction de la même façon qu'avec ADDA. ADDB permet d'ajouter les valeurs inscrites dans le registre B et dans un octet mémoire. Le résultat de l'addition se trouve dans B.

ADDD

Il s'agit là encore d'ajouter le contenu d'un registre à une valeur prise en mémoire. D étant un registre 16 bits, on pourra donc additionner deux valeurs elles-mêmes de 16 bits. Modes d'adressage possibles : immédiat, indexé et étendu.

Exemple : MODE D'ADRESSAGE ÉTENDU (10 octets)

1. Programme BASIC

```
10
20
30 INPUT "PREMIER NOMBRE"; N
40 N1 = INT(N/256) : POKE 32750,N1
50 N2 = N-N1*256 : POKE 32751,N2
60 INPUT "DEUXIEME NOMBRE";M
70 M1 = INT(M/256) : POKE 32752,M1
80 M2 = M-M1*256 : POKE 32753,M2
90 EXEC 32701
100 PRINT "LA SOMME VAUT";256*PEEK(32754) + PEEK(32755)
```

2. Programme assembleur

Lignes	Codes machine	Assembleur	
1	FC 7F EE	LDD	>32750
2	F3 7F F0	ADDD	>32752
3	FD 7F F2	STD	>32754
4	39	RTS	

Ligne 1 : le registre D est chargé avec le premier nombre N. Tenant compte du fait que D est un registre 16 bits, l'instruction LDD va chercher en mémoire la valeur des 2 octets 32750 et 32751. Regardons, en supposant N égal à 1000, comment cela se passe.

Ligne 3 : on retranche à N le nombre M contenu dans l'octet 32715(3 + X). Le nombre M est connu du programme dès que la ligne BASIC 40 est exécutée.

Notons que la soustraction se fait toujours dans le même sens : c'est l'octet mémoire qui est retranché à l'accumulateur et non le contraire.

Ligne 4 : puisque le résultat de l'opération est maintenant dans A, il ne reste plus qu'à stocker ce résultat dans un emplacement mémoire déterminé que le programme BASIC pourra retrouver. La ligne 60 commandant d'aller chercher la réponse dans l'octet 32760, il nous faut comprendre en quoi STA [,X] a un rapport avec l'octet en question. C'est le premier programme dans lequel est utilisé le mode d'adressage indirect indexé, aussi voyons la différence avec l'adressage indexé habituel :

```
STA ,X
```

si cette instruction avait été écrite à la ligne 4, le résultat de la différence aurait été rangé dans l'octet pointé par X, donc à l'adresse 32712.

```
STA [,X]
```

le résultat est en réalité placé dans l'octet dont l'adresse est donnée par les 16 bits des octets 32712 et 32713.

Reprenons : X contient le nombre 32712, l'octet 32712 contient le nombre 7F (127 en décimal), et le 32713 contient le nombre F8 (248 en décimal), ce qui nous donne, en appliquant la règle poids fort-poids faible, l'adresse définitive de l'octet dans lequel sera rangé le résultat :

$$127*256 + 248 = 32760$$

Avant de passer aux instructions suivantes, ne manquez pas de faire tourner ce programme en lui proposant des calculs du genre 10-11 ou 0-255 et en analysant les réponses de l'ordinateur.

SUBB SUBD

L'instruction SUBB est utilisable de la même façon que SUBA. L'instruction SUBD permet quant à elle d'effectuer des soustractions 16 bits.

MUL

Cette instruction, n'acceptant que le seul mode d'adressage inhérent, effectue le produit de deux valeurs 8 bits contenues dans les accumulateurs A et B. Le résultat de la multiplication, sur 16 bits, est inscrit dans le registre D faisant ainsi disparaître les valeurs initiales de A et B.

Exemple : MODE D'ADRESSAGE INHÉRENT (20 octets)

1. Programme BASIC

```
10
20
30 INPUT "DONNEZ UN NOMBRE";N:POKE 32750,N
40 EXEC 32701
50 PRINT "EN VOICI SON DOUBLE";256*PEEK(32751) +
   PEEK(32752)
60 PRINT "EN VOICI SON CARRE";256*PEEK(32753) +
   PEEK(32754) :GOTO 30
```

2. Programme assembleur

Lignes	Codes machine	Assembleur	
1	B6 7F EE	LDA	> 32750
2	C6 02	LDB	# 2
3	3D	MUL	
4	FD 7F EF	STD	> 32751
5	B6 7F EE	LDA	> 32750
6	F6 7F EE	LDB	> 32750
7	3D	MUL	
8	FD 7F F1	STD	> 32753
9	39	RTS	

C'est la ligne BASIC 30 qui a chargé par POKE dans l'octet 32750 le nombre N dont on va calculer le double puis le carré. N est un nom-

bre de 8 bits et il n'est pas possible de proposer un nombre supérieur à 255 sans que l'ordinateur ne renvoie le message FC ERROR.

Lignes 1 et 2 : le registre A va contenir la valeur de N et le chiffre 2 est placé dans B.

Ligne 3 : les valeurs des deux accumulateurs sont multipliées, ce qui revient pour nous à calculer le double du nombre qui a été donné au départ. Le résultat sur 16 bits de cette opération se retrouve alors dans D et naturellement les valeurs qui étaient dans A et B sont perdues car, ne l'oublions pas, A et B constituent les registres de poids fort et de poids faible de D.

Ligne 4 : le résultat de la multiplication est rangé en mémoire dans les octets 32751 et 32752 ; la ligne 50 retrouve pour nous la valeur décimale de ce résultat non sans avoir, mais c'est déjà de l'histoire ancienne, multiplié le contenu de l'octet fort par 256.

Lignes 5 et 6 : on réintroduit dans A et B les nombres nécessaires à la suite du programme : puisqu'il s'agira de calculer un carré les deux registres doivent contenir la même valeur, en l'occurrence N.

Lignes 7 et 8 : le produit de N par lui-même est calculé, le résultat placé dans D et son contenu rangé dans les octets 32753 et 32754, octets que le programme BASIC retrouvera à la ligne 60.

Il n'y a rien d'autre à ajouter concernant l'instruction MUL.

Avant de passer aux instructions de branchement, pourquoi ne pas récrire le même programme en le débutant par LDX#32750 et en utilisant le plus souvent possible l'adressage indexé. Le lecteur ne devra jamais perdre de vue que si le mode indexé est un peu moins lisible que l'étendu, il a en revanche une qualité non négligeable : il est bien souvent moins gourmand en octets que ce dernier.

BEQ BNE BRA BSR

Tous les programmes que nous avons étudiés jusqu'à maintenant étaient conçus sur le type séquentiel, ce qui veut dire que les instructions étaient exécutées les unes à la suite des autres, dans l'ordre où elles avaient été écrites. Nous savons tous qu'en BASIC il est possible, avec des instructions comme GOTO par exemple, d'empêcher le programme de se dérouler normalement en l'obligeant à se brancher à un numéro de ligne choisi par nous. Voyons comment nous devrons nous y prendre pour obtenir le même effet. Nous retrouverons, après quelques explications théoriques, l'étude d'exemples bien concrets.

BEQ

Branch if Equal

Branchement si égal

Le branchement à l'une des parties du programme machine ne se fera que si l'une des deux conditions suivantes vient d'être réalisée :

1 — soustraction entre deux nombres égaux.

2 — comparaison entre deux nombres égaux.

C'est l'octet qui suit immédiatement l'instruction BEQ qui, en mode complément à 2, indiquera alors au processeur quelle autre partie du programme devra être exécutée.

Dans le cas d'une comparaison ou d'une soustraction entre deux nombres différents, BEQ n'aura aucun effet et c'est l'instruction écrite à la ligne d'après qui sera exécutée.

Pour résumer, disons que l'instruction BEQ s'utilise de la même façon que la phrase BASIC bien connue :

```
IF A = B THEN...
```

BNE

Branch if Not Equal
Branchement si non égal

Voici l'instruction contraire de la précédente. Cette fois le branchement ne sera effectué que dans le cas où l'une des deux situations suivantes se sera présentée :

1 — soustraction entre deux nombres différents.
2 — comparaison effectuée sur deux nombres différents.

Ici aussi l'endroit du programme où le branchement devra se faire sera indiqué par l'octet placé après l'instruction BNE. Le nombre écrit dans cet octet devra l'être sous la forme complément à deux.

On peut trouver l'équivalent BASIC de BNE en écrivant :

```
IF A < > B THEN...
```

BRA

Branch Always
Branchement dans tous les cas

Le branchement à la partie du programme indiquée par l'octet qui suit l'instruction BRA est un branchement inconditionnel. Ce type de branchement ne se préoccupe pas de savoir si telle ou telle condition a été réalisée : il s'effectue de toute manière.

Vous aurez reconnu en BRA l'équivalent assembleur de la commande BASIC GOTO.

BSR

Branch at Subroutine
Branchement vers un sous-programme

Après GOTO, voici GOSUB : BSR est en effet l'instruction de branchement qui permet de sauter jusqu'à un sous-programme. Il s'agit

comme pour BRA d'un branchement inconditionnel qui s'effectuera dans tous les cas.

Il est inconcevable, en BASIC, d'écrire un GOSUB sans prévoir le RETURN qui nous ramènera au programme principal.

Il en est de même en assembleur et il nous faudra toujours penser à terminer nos sous-programmes par une instruction que nous avons rencontrée dès notre premier exemple : RTS (*Return from subroutine*).

CLRA

Le registre A est mis à zéro, ce qui revient à dire que A est chargé avec la valeur 0. Il n'y a qu'une seule possibilité pour l'adressage : le mode inhérent.

Exemple : MODE D'ADRESSAGE INHÉRENT (13 octets)

1. Programme BASIC

```
10
20
30 GOSUB 40:EXEC 32701:PRINT:GOSUB 40:END
40 FOR I = 32740 TO 32749 :PRINT PEEK(I);
50 NEXT : RETURN
```

2. Programme assembleur

Lignes	Codes machine	Assembleur		
1	4F		CLRA	
2	C6 0A		LDB	#10
3	8E 7F E4		LDX	#32740
4	A7 80	SUITE	STA	,X+
5	CO 01		SUBB	#1
6	26 FA		BNE	SUITE(−6)
7	39		RTS	

Le but de ce programme est d'écrire la valeur 0 dans chacun des 10 octets numérotés de 32740 à 32749. Le programme BASIC fait apparaître sur l'écran ce que contiennent ces octets avant et après exécution des lignes écrites en langage machine.

Ligne 1 : les explications concernant l'instruction CLRA tiennent en quelques mots : le chiffre 0 est écrit dans A et l'on aurait pu tout aussi bien, avec un octet de plus il est vrai, écrire LDA#0.

Ligne 2 : B est chargé avec 10. Le registre nous servira de compteur pour la boucle qui sera exécutée 10 fois.

Ligne 3 : X contient au début du programme la valeur 32740 et de ce fait X pointe le premier octet que nous aurons à rendre nul.

Ligne 4 : on range la valeur de A dans l'octet 32740. Comme A a été mis à zéro à la ligne 1, cet octet contient donc maintenant le nombre 0. Dans cette ligne 4, il faut observer que le registre X est incrémenté et pointe alors vers 32741.

Ligne 5 : on retranche 1 à B, celui-ci vaut alors 9.

Ligne 6 : puisque la soustraction a été effectuée entre deux valeurs différentes (10 et 1), l'instruction BNE branchera le programme à la ligne 4. L'octet 32741 sera alors mis à zéro, le registre X, incrémenté, pointera vers 32742. La ligne 5 retranchera alors 1 de B qui, à ce moment-là, vaudra 8. Le test de branchement de la ligne 6 renverra à nouveau le programme en ligne 4.

Nous avons donc une boucle qui sera exécutée 10 fois. Après avoir rendu nul le dixième octet (32749), la ligne 5 effectuera la différence entre le contenu de B (qui vaudra alors 1) et le nombre 1. Puisque cette différence ne sera plus différente de 0, la ligne 6 n'aura plus aucun effet et il ne restera plus à la ligne 7 qu'à retourner au programme BASIC.

Le lecteur doit s'habituer à la façon dont est présentée la partie assembleur. La ligne 3 a été appelée SUITE et du coup la ligne 5 s'écrit BNE SUITE au lieu de BNE ligne 3. BNE SUITE a été codée 26 FA.FA est en complément à 2 le nombre -6. On indique ainsi au microprocesseur qu'il doit repartir en arrière de 6 octets, le décompte s'effectuant toujours à partir du premier octet de la ligne qui suit l'instruction de branchement.

CLRB

Cette instruction se programme de la même manière que CLRA. L'adressage inhérent est le seul accepté.

CLR ,X

Le contenu de l'octet mémoire spécifié est rendu nul. On peut utiliser les modes d'adressage indexé et étendu.

Exemple : MODE D'ADRESSAGE INDEXÉ (14 octets)

1. Programme BASIC

```
10
20
30 DIM A%(999) : FOR I = 0 TO 999
40 A%(I) = 1:NEXT:BEEP
50 U = VARPTR(A%(0)) :POKE 32715,INT(U/256)
60 POKE 32716,U-INT(U/256)*256
70 EXEC 32701
80 FOR I = 0 TO 999 : PRINT A%(I);:NEXT
```

2. Programme assembleur

Lignes	Codes machine	Assembleur		
1	BE 7F CB		LDX	> 32715
2	CC 07 DO		LDD	#2000
3	6F 80	SUITE	CLR	,X+
4	83 00 01		SUBD	#1
5	26 F9		BNE	SUITE(−7)
6	39		RTS	

L'assembleur, pour une fois, se comprend mieux que le programme BASIC dans lequel il est incorporé.

Ligne 1 : supposons qu'il y ait les valeurs 102 dans l'octet de poids fort 32715 et 245 dans l'octet de poids faible 32716. Le registre X, de 16 bits ne le perdons pas de vue, contiendra alors le nombre

26357 (102*256+245).

Ligne 2 : D contient au départ le nombre 2000. Nous devinons déjà une structure de boucle qui sera exécutée 2000 fois.

Ligne 3 : l'octet pointé par X, donc l'octet 26357 dans notre exemple, est mis à zéro et, juste après, la valeur de X passe à 26358.

Lignes 4 et 5 : on retranche 1 de D qui passe alors à 1999 puis l'instruction BNE nous renvoie 7 octets en arrière, c'est-à-dire à la ligne SUITE.

Nous avons donc une boucle qui sera effectuée 2000 fois, rendant ainsi nuls les contenus des octets dont les adresses s'échelonnent de 26357 à 28356. Pour savoir à quoi peut bien nous servir la mise à zéro de toute cette zone mémoire il nous faut maintenant revenir aux lignes BASIC. Les variables entières sont rangées en mémoire sur deux octets à une adresse qui nous est donnée par la fonction VARPTR.

Essayons en mode direct :

```
Z% = 260 :U = VARPTR(Z%)
PRINT U:PRINT PEEK(U):PRINT PEEK(U+1)
```

Admettons que l'ordinateur réponde :

```
U = 26357 :PEEK(U)=1:PEEK(U+1)=4
```

Cela voudra dire qu'à l'adresse 26357 sera écrite la partie forte de la variable entière Z% et que la partie faible de cette même variable sera à 26358.

```
Z% =260=256*1+4
```

Toujours en mode direct, tapons la ligne suivante :

```
POKE 26357,0 :POKE 26358,0 :PRINT Z%
```

La réponse est sans ambiguïté, Z% est alors égal à 0 (256*0+0)

Venons-en maintenant à notre programme et à nos 1000 variables du tableau A%. Supposons ici aussi que U soit égal à 26357 et annulons ensuite les octets 26357 et 26358:A%(0) vaudra alors 0.

Annulons ensuite les octets 26359 et 26360 ; A%(1) vaudra alors aussi 0. Quand on aura initialisé à 0, avec le programme assembleur, les contenus des 2000 octets placés à partir de l'adresse 26357, nous aurons donc annulé les 1000 variables entières A% et la ligne BASIC 80 n'aura plus qu'à le confirmer.

INCA

Cette instruction incrémente le registre A, c'est-à-dire qu'elle lui ajoute une unité. Un seul mode d'adressage possible : l'inhérent.

Exemple : MODE D'ADRESSAGE INHÉRENT (14 octets)

1. Programme BASIC

```
10
20
30 EXEC 32701
```

2. Programme assembleur

Lignes	Codes machine	Assembleur		
1	4F		CLRA	
2	C6 FF		LDB	#255
3	8E 40 00		LDX	#16384
4	E7 80	SUITE	STB	,X+
5	4C		INCA	
6	80 00		SUBA	#0
7	26 F9		BNE	SUITE (−7)
8	39		RTS	

Lignes 1, 2 et 3 : les valeurs 0, 255 et 16384 sont chargées respectivement dans les registres A, B et X. Rappelons que 16384 est l'adresse du premier octet de la mémoire vidéo.

Ligne 4 : le nombre 255 est rangé dans l'octet pointé par X, c'est-à-dire l'octet 16384. Comme cette valeur correspond au premier segment en haut à gauche de l'écran, celui-ci voit ses huit bits s'allumer. Dès que ceci est exécuté, le registre X passe à 16385.

Ligne 5 : INCA ajoute 1 au contenu de l'accumulateur qui prend donc la valeur 1. Cette ligne est équivalente à la ligne ADDA #1 mais elle n'utilise qu'un seul code machine au lieu de deux.

Ligne 6 : on retranche au registre A le nombre 0. A garde la même valeur 1, la soustraction de 0 à n'importe quel nombre ne modifiant jamais ce dernier. Voici une instruction qui serait tout à fait inutile si elle n'allait nous servir à la ligne suivante.

Ligne 7 : le branchement sept octets en arrière s'effectuera si la différence précédente a porté sur deux nombres non égaux. Ce qui est le cas, on a retranché 0 à 1.

On retrouve alors la ligne 4 qui va permettre cette fois d'allumer le deuxième segment de l'écran, celui qui correspond à l'octet 16385. Le registre X, incrémenté, contiendra alors le nombre 16386. Puis — lignes 5 à 7 —, une unité sera ajoutée à A, et 0 sera retranché à la nouvelle valeur (2) de ce registre ; cette soustraction se faisant sur deux chiffres différents, un branchement sera à nouveau effectué à la ligne 4.

Le but de cette boucle est donc d'allumer à chaque passage un segment de plus sur notre écran et c'est effectivement ce que l'exécution du programme BASIC nous montre : une bande de couleur apparaît en haut du téléviseur.

Essayons, pour finir, de savoir à quel moment l'ordinateur sortira de la boucle. Le registre A étant augmenté de 1 à chaque fois, finira par arriver à 255 (ses huit bits à 1). L'incrémentation suivante ramènera tous ses bits à zéro et la soustraction de la ligne 6 portera alors sur deux nombres égaux. A ce moment-là, l'instruction BNE sera devenue sans effet et c'est l'ordre d'après, RTS, qui s'exécutera.

INCB

Cette instruction augmente d'une unité le contenu du registre B. INCB ne peut être utilisée qu'avec le mode d'adressage inhérent.

INC

Avec INC, la possibilité est donnée d'incrémenter — ajouter 1 — au contenu d'un octet de la mémoire. L'adressage indexé et l'adressage étendu sont les deux modes possibles.

Exemple : MODE D'ADRESSAGE ÉTENDU (13 octets)

1. Programme BASIC

```
10
20
30 INPUT "PREMIER NOMBRE" ; N : POKE 32750 , N
40 INPUT "DEUXIEME NOMBRE" ; M : POKE 32751 , M+1
50 EXEC 32701
60 PRINT "LA SOMME VAUT" ; PEEK(32750) : GOTO 30
```

2. Programme assembleur

Lignes	Codes machine	Assembleur		
1	F6 7F EF		LDB	>32751
2	C0 01	SUITE	SUBB	#1
3	27 05		BEQ	FIN(+5)
4	7C 7F EE		INC	>32750
5	20 F7		BRA	SUITE(−9)
6	39		RTS	

La lecture des lignes BASIC indique qu'il s'agit là d'un programme d'addition. L'instruction ADD n'apparaît pas dans la partie assembleur car elle a été remplacée par une boucle incrémentant le contenu d'un octet et ceci le nombre de fois voulu.

Ligne 1 : le contenu de l'octet 32751 est placé dans l'accumulateur B. Ce qui fait que B contient le nombre M+1. Cette valeur est égale au second terme de la somme augmenté de 1.

Ligne 2 : on retranche 1 au registre B. Voici que s'explique la raison pour laquelle on est parti d'une valeur supérieure d'une unité pour B, ceci compense cela.

Ligne 3 : si la différence porte sur deux chiffres égaux, c'est-à-dire si B vaut 1, un branchement est effectué à la ligne 6, ligne appelée FIN. 27 est le code machine de l'instruction BEQ et 05 est le nombre d'octets que le programme va devoir sauter. Vous rappelez-vous le registre PC, le compteur ordinal ? C'est lui qui va réaliser ce branchement. Il contient toujours l'adresse de la prochaine instruction à exécuter donc, dans notre exemple, il est chargé avec le nombre 32708 (faites le compte, le code de l'instruction INC est bien dans cet octet). Ainsi, lorsque le test de la ligne 3 indiquera qu'un branchement est nécessaire, 5 unités seront ajoutées au registre PC qui ira pointer sur l'octet 32713, octet correspondant à RTS. Retenez de tout ceci que le décompte du nombre d'octets dans les opérations de branchement se fait toujours à partir du début de la ligne suivante.

Ligne 4 : on ajoute 1 au contenu de l'octet 32750, donc au premier terme de la somme.

Ligne 5 : BRA est l'instruction de branchement inconditionnel. Equivalente de GOTO, elle renvoie le processeur à la ligne 2 pour la suite du programme.

Nous nous retrouvons une fois de plus devant une boucle qui, à chaque passage, procède aux deux opérations suivantes :

— 1 est enlevé au registre B, c'est-à-dire au deuxième nombre de la somme.
— 1 est ajouté à l'octet 32750, c'est-à-dire au premier terme de la somme.

B étant décrémenté à chaque fois, sa valeur arrivera forcément à 1. La ligne assembleur 2 effectuera alors une différence donnant un résultat nul, et le branchement BEQ sera réalisé. L'octet 32750 contiendra à la fin du programme la somme des deux nombres que nous avons proposés à l'ordinateur.

Reprenons rapidement ce que nous avons déjà dit à propos des additions sur huit bits (voir l'instruction ADDA) : lorsque le résultat d'une somme arrive à 256, il est ramené à 0.

Exemple : 200 + 100 = 256 + 44 = 44.

DECA

Le contenu de l'accumulateur A est décrémenté, c'est-à-dire qu'une unité lui est retirée. Le mode d'adressage inhérent est le seul utilisable.

Exemple : MODE D'ADRESSAGE INHÉRENT (13 octets)

1. Programme BASIC

```
 5 CLS : LOCATE 0, 10
10
20
30 EXEC 32701
```

2. Programme assembleur

Lignes	Codes machine	Assembleur		
1	C6 55		LDB	#85
2	86 F0		LDA	#240
3	8E 40 00		LDX	#16384
4	E7 81	SUITE	STB	,X+ +
5	4A		DECA	
6	26 FB		BNE	SUITE(−5)
7	39		RTS	

L'instruction DECA est très utilisée en assembleur pour servir de compteur dans une boucle.

Lignes 1, 2 et 3 : dans notre exemple, on veut qu'une action se reproduise 240 fois et ce nombre est placé dans le registre A. Remarquons que X contient l'adresse du premier octet de la mémoire écran et que B est chargé avec un nombre qui s'écrit 01010101 en binaire.

Ligne 4 : on inscrit dans l'octet pointé par X le nombre 85. Puisque X vaut 16384 et que 85 s'écrit en binaire sous la forme d'une suite constituée alternativement des chiffres 0 et 1, on peut prévoir l'effet de cette instruction sur notre écran : le premier segment en haut et à gauche s'allume en pointillés. Le registre X est alors incrémenté deux fois et prend donc la valeur 16386 : ceci se repère par les deux signes + placés à la suite de X.

Ligne 5 : l'accumulateur A passe de la valeur 240 à la valeur 239 ; on aurait obtenu le même résultat avec SUBA #1.

Ligne 6 : puisque DECA est une différence entre A et 1, le branchement BNE à la ligne 4 se réalisera tant que A sera différent de 1. Nous en sommes pour l'instant à 239 pour ce registre, il faut donc repartir dans la boucle.

Le registre B n'ayant pas varié, c'est à nouveau le nombre 85 qui sera écrit dans l'octet pointé par X. Ce programme ne va donc concerner qu'un segment sur deux puisque c'est le numéro 3 qui s'allumera en pointillés. Ensuite A se verra décrémenté et prendra la valeur 238.

Nous avons affaire ici à une boucle qui sera exécutée 240 fois, elle allume au passage 240 segments sur l'écran. Comme les adresses des segments allumés vont de deux en deux, seuls 20 segments apparaîtront sur une ligne d'écran et douze lignes, alternant pointillés et cases vides, seront tracées devant nos yeux.

Ce programme assembleur ressemble beaucoup à celui rencontré avec l'instruction INCA. Afin de faire le point sur ses connaissances, le lecteur pourra peut-être reprendre le programme INCA et le modifier pour qu'il puisse tourner avec DEC ; et par la même occasion changer les registres utilisés, Y à la place de X et B à la place de A par exemple.

DECB

Cette instruction procède avec l'accumulateur B de la même façon que DECA le fait avec A. Elle retranche 1 du contenu de B. Le mode d'adressage utilisable est l'inhérent.

DEC

Cette instruction permet de retirer 1 de la valeur d'un octet en mémoire. On peut se servir des modes d'adressage indexé et étendu.

Exemple : MODE D'ADRESSAGE ÉTENDU (14 octets)

1. Programme BASIC

```
10
20
30 CLS : ATTRB 1 , 1 : LOCATE 10, 10
40 PRINT "DEBUT" : PLAY "L5DOREMI" : EXEC 32701
50 LOCATE 10 , 10 : PRINT "FIN" : PLAY "SILASO"
```

2. Programme assembleur

Lignes	Codes machine	Assembleur		
1	86 FF		LDA	# 255
2	B7 7F EE		STA	> 32750
3	4A	SUITE	DECA	
4	26 FD		BNE	SUITE(−3)
5	7A 7F EE		DEC	> 32750
6	26 F8		BNE	SUITE(−8)
7	39		RTS	

Voici le programme qui réalise une boucle de temporisation. Elle est équivalente en durée à la bouche BASIC :

```
FOR I = 1 TO 300 : NEXT
```

Mais si le BASIC n'a rien fait d'autre que de compter jusqu'à 300, l'assembleur, pendant le même temps, a eu le temps d'exécuter sa boucle vide plus de 60 000 fois. Ceci met en avant l'extraordinaire rapidité des programmes écrits en langage machine.

Lignes 1 et 2 : le registre A est chargé avec la valeur maximum et cette valeur est écrite dans l'octet 32750.

Lignes 3 et 4 : A valant 255, la décrémentation lui soustrait une unité et le fait passer à 254. On doit être habitué maintenant à l'instruction BNE qui va renvoyer le programme à la ligne 3. Ainsi le microprocesseur n'a effectué aucune action visible : il n'a fait que perdre du temps. La ligne 3 place dans A le nombre 253 et à nouveau le branchement BNE fait retourner à la ligne SUITE. On retrouve donc, avec ces deux lignes, mais en beaucoup plus rapide, la ligne BASIC :

```
FOR I = 254 TO 0 : NEXT
```

Ligne 5 : l'accumulateur étant alors à zéro, c'est au tour de l'octet 32750 d'être décrémenté. Il avait été chargé au départ avec le nombre 255, il va donc contenir 254.

Ligne 6 : nouvelle utilisation de BNE qui concernera la dernière soustraction effectuée, en l'occurrence la décrémentation de la ligne 5. Puisque cette différence aura été faite entre les nombres différents 255 et 1, le programme se rebranchera à SUITE dont huit octets en arrière.

On retrouve alors la ligne 3. A sera une nouvelle fois décrémenté et, partant de 0, repassera à 255. N'oublions pas que −1 s'écrit pour le processeur 255 (ou FF hexadécimal). Nous revoilà de nouveau dans une boucle qui va faire passer A de 255 à 0 (lignes 3 et 4), puis à terme, une décrémentation de l'octet 32750 (ligne 5) sera réalisée. Puisque cet octet en sera alors à la valeur 253, il y aura encore branchement à la ligne 3.

Le principe doit être dès lors compris : tant que le contenu de l'octet 32750 ne sera pas nul, le programme bouclera.

Vous avez certainement remarqué, en exécutant le programme, qu'à peine une demi-seconde s'écoulait entre les affichages des mots DEBUT et FIN sur l'écran. Essayez d'utiliser le principe de la bouche d'attente qui vient d'être étudié pour obtenir une temporisation plus grande. Un conseil : utilisez un nouvel octet que vous décrémenterez régulièrement à chaque fois que l'octet 32750 sera arrivé à zéro.

BHI BLO BHS BLS

Nous abordons maintenant quatre nouvelles instructions de branchement qui vont pouvoir, lorsque les conditions voulues seront réalisées, mettre une nouvelle valeur dans le registre PC et permettre ainsi d'annuler le déroulement séquentiel du programme. Ces instructions vont porter sur la comparaison des grandeurs de deux nombres, dont le premier sera toujours dans un registre. Rappelons que pour pouvoir utiliser BEQ et BNE, il fallait qu'une soustraction ou une comparaison ait été effectuée auparavant. Il va en être de même pour ces quatre nouvelles instructions.

BHI

Branch on Higher

Branchement si plus grand

BHI réalisera un branchement à l'un des octets du programme machine dans l'un des deux cas suivants :

1 — soustraction entre deux nombres dont le premier, celui contenu dans le registre, est strictement supérieur au deuxième.

2 — comparaison entre deux nombres dont le premier, là encore contenu dans le registre, est strictement supérieur au second.

L'octet suivant l'instruction BHI précisera, sur le mode complément à deux, quelle partie du programme devra alors être exécutée. Naturellement, cette instruction sera sans effet si le registre est inférieur ou égal à l'autre nombre.

A noter que la comparaison se fera sans que l'ordinateur tienne compte de la valeur en complément à deux de ces nombres. Si, par exemple, les deux nombres valent 254 et 10, le premier sera considéré comme supérieur au deuxième bien que ce soit en réalité le nombre −2 en complément à deux.

Si l'on veut trouver l'équivalent BASIC de BHI, on doit écrire :

```
IF A > B THEN ...
```

BLO

Branch on Lower

Branchement si plus petit

Cette fois le branchement s'effectuera dans l'un des cas suivants :

1 — soustraction entre deux nombres dont le premier est strictement inférieur au deuxième.

2 — comparaison entre deux nombres dont le premier est strictement inférieur au deuxième.

Le premier nombre étant toujours celui qui est dans le registre.

Pour cette instruction aussi, l'ordinateur ne tiendra pas compte des valeurs négatives, celles qui correspondent au complément à deux. Ecrivons la ligne BASIC correspondante :

```
IF A < B THEN ...
```

BHS

Branch on Higher or Same

Branchement si supérieur ou égal

Le branchement sera exécuté si l'une des deux conditions suivantes est vraie :

1 — soustraction entre deux nombres dont le premier est supérieur ou égal au deuxième.

2 — comparaison entre deux nombres dont le premier est supérieur ou égal au deuxième.

Là encore, d'une part le premier nombre se trouve dans le registre considéré, d'autre part on ne tient pas compte de la valeur en complément à deux. Voici comment, en BASIC, on écrirait cette instruction :

```
IF A > = B THEN ...
```

BLS

Branch on Low or Same

Branchement si inférieur ou égal

Le branchement sera effectué si l'une des deux conditions suivantes est réalisée :

1 — soustraction entre deux nombres dont le premier est inférieur ou égal au deuxième.

2 — comparaison entre deux nombres dont le premier est inférieur ou égal au deuxième.

Cette dernière instruction respecte les mêmes règles que les précédentes : le premier terme de la différence ou de la comparaison provient du registre et le mode complément à deux n'est pas pris en compte. Donnons la traduction BASIC :

```
IF A < = B THEN ...
```

CMPA

Une comparaison sera réalisée avec le nombre placé immédiatement après cette instruction. Une instruction de branchement doit suivre normalement cette comparaison. Les modes d'adressage immédiat, indexé et étendu peuvent être utilisés.

Exemple : MODE D'ADRESSAGE ÉTENDU (24 octets)

1. Programme BASIC

```
10
20
30 X = INT (RND*256) : POKE 32750 , X
40 INPUT "QUEL NOMBRE PROPOSEZ-VOUS" ;
   N : POKE 32751 , N
50 EXEC 32701 : ON PEEK (32752) GOTO 60, 70, 80
60 PRINT "VOUS AVEZ GAGNE" : END
70 PRINT "TROP GRAND !" : GOTO 40
80 PRINT "TROP PETIT !" : GOTO 40
```

2. Programme assembleur

Lignes	Codes machine	Assembleur		
1	B6 7F EF		LDA	>32751
2	B1 7F EE		CMPA	>32750
3	27 06		BEQ	EGAL(+6)
4	22 08		BHI	SUP (+8)
5	86 03	INF	LDA	# 3
6	20 06		BRA	FIN (+6)
7	86 01	EGAL	LDA	# 1
8	20 02		BRA	FIN (+2)
9	86 02	SUP	LDA	# 2
10	B7 7F F0	FIN	STA	>32752
11	39		RTS	

Voici une version du jeu qui consiste à deviner un nombre que l'ordinateur aura choisi. Le branchement ON GOTO de la ligne BASIC 50 s'effectuera en fonction du nombre trouvé dans l'octet 32752. Voyons comment l'assembleur y aura placé la valeur correcte 1, 2 ou 3.

Ligne 1 : la ligne BASIC 40 aura écrit dans l'octet 32751 le nombre N proposé. C'est donc le registre A qui va contenir ce nombre.

Ligne 2 : une comparaison est effectuée entre le nombre proposé et le contenu de l'octet 32750. Or dans cet octet, a été inscrit par POKE le nombre X que l'ordinateur a tiré au hasard. Voici donc la ligne qui va réaliser la comparaison sur laquelle est basée tout le programme.

Ligne 3 : si la comparaison a porté sur deux nombres égaux, cela voudra dire que l'on a gagné. BEQ va procéder alors à un branchement vers la ligne 7, six octets plus loin. LDA #1 va, à ce moment-là, placer dans l'accumulateur le nombre 1 et un branchement inconditionnel (ligne 8) va entraîner le processeur à l'avant-dernière ligne du programme. Il ne restera plus alors qu'à écrire la valeur 1 dans l'octet 32752. Le BASIC retrouvera ce nombre et le branchement ON GOTO fera imprimer le message "VOUS AVEZ GAGNE".

Ligne 4 : si l'on suppose maintenant que A est supérieur au nombre choisi par l'ordinateur, l'instruction BHI nous conduira à la ligne 9. Le registre A sera chargé avec la valeur 2, valeur qui sera ensuite écrite (ligne 10) dans l'octet 32752. Il ne restera plus au BASIC qu'à retrouver le contenu de cet octet et le message "TROP GRAND" sera affiché sur l'écran.

Lignes 5 et 6 : dernière possibilité enfin, on a proposé à l'ordinateur un nombre trop petit. Les instructions BEQ et BHI sont restées sans effet et le programme s'est déroulé séquentiellement jusqu'à ces lignes. C'est le chiffre 3 qui sera écrit dans l'accumulateur avant qu'un branchement inconditionnel n'envoie le microprocesseur à la ligne 10. 3 est alors recopié dans l'octet 32752 et l'instruction BASIC ON GOTO donnera la réponse à notre essai : "TROP PETIT".

CMPB

D'une manière identique à CMPA, cette instruction compare le nombre écrit dans le registre B avec le nombre placé immédiatement après. On peut se servir des modes d'adressage immédiat, étendu et indexé.

CMPX

CMPX établit une comparaison entre deux nombres de 16 bits. Une instruction de branchement doit ensuite exploiter cette comparaison. On peut utiliser les modes d'adressage immédiat, indexé et étendu.

Exemple : MODE D'ADRESSAGE INDEXÉ (26 octets)

1. Programme BASIC

```
10
20
30 DIMA %(99) : FOR I = 0 TO 99 : A%(I) = INT(RND*10000)
   +1 : NEXT
40 U = VARPTR(A% (0)) : POKE 32750, INT(U/256)
50 POKE 32751, U-INT(U/256)*256
60 BEEP : EXEC 32701
70 PRINT "LE PLUS GRAND NOMBRE DU TABLEAU EST" ;
   256*PEEK (32752) + PEEK (32753)
```

2. Programme assembleur

Lignes	Codes machine	Assembleur		
1	8E 00 00		LDX	#0
2	86 65		LDA	#101
3	10 BE 7F EE		LDY	>32750
4	4A	SUITE	DECA	
5	27 0D		BEQ	FIN(+13)
6	AC A1		CMPX	,Y++
7	24 F9		BHS	SUITE(-7)
8	AE A3		LDX	,--Y
9	BF 7F F0		STX	>32752
10	EE A1		LDU	,Y++
11	20 F0		BRA	SUITE(-16)
12	39	FIN	RTS	

Le but de ce programme est de rechercher dans une liste de cent nombres compris entre 1 et 10000, l'élément maximum. Ces cent valeurs ont été choisies de façon aléatoire et sont placées dans un tableau entier A%. L'exécution du programme machine peut être considérée comme instantanée, ce qui ne serait pas le cas d'une version BASIC écrite avec un test du type IF THEN.

Lignes 1, 2 et 3 : c'est l'initialisation du programme assembleur. A est le nombre d'éléments du tableau, nombre que l'on a pris la précaution d'augmenter d'une unité à cause de DECA qui va intervenir dès le début de la boucle. X est mis à zéro, ce sera à la fin du programme l'élément maximum. Quant au registre 16 bits Y, il est chargé à l'aide des valeurs écrites dans les octets 32750 et 32751. Comme les instructions BASIC 40 et 50 ont placé dans ces deux octets le nombre VARPTR (A%(0)), Y contient donc l'adresse du premier octet du tableau. A titre d'exemple, supposons que VARPTR(A%(0)) soit égal à 26429 : cela voudra dire que l'entier A%(0) est écrit en mémoire sur les deux octets 26429 et 26430, poids fort sur le premier et poids faible sur l'autre. Bien entendu A%(1) sera placé sur les octets 26431 et 26432, et ceci se poursuivra jusqu'à A%(99).

Lignes 4 et 5 : l'accumulateur ne sert que de compteur : il est décrémenté et un branchement à la fin du programme sera réalisé dès que les cent nombres auront été passés en revue.

Lignes 6 et 7 : une comparaison est effectuée entre le contenu du registre X, 0 en début de programme, et le nombre de 16 bits pointé par Y, c'est-à-dire A%(0). Puisque X est plus petit que A%(0), l'instruction BHS est sans effet et le programme se poursuit séquentiellement. Notons que Y a été augmenté de deux et vaut alors 26431.

Ligne 8 : X doit être à la fin de la boucle le plus grand élément du tableau ; à chaque fois qu'il sera dépassé par un nombre, il devra prendre la valeur de ce dernier. C'est ce qui se passe ici, X se trouve chargé avec A%(0). Les deux signes – écrits devant Y précisent qu'Y avait été ramené à 26439 avant cette opération.

Ligne 9 : on range dans un emplacement mémoire la nouvelle valeur de X et c'est là que le BASIC viendra puiser la réponse au problème.

Lignes 10 et 11 : le registre U n'a rien à voir avec ce programme, c'est simplement un artifice destiné à refaire passer la valeur de Y à 26431 avant de retourner à la ligne 4 pour un second passage.

En définitive, le cœur du problème se trouve placé aux lignes 6 et 7 qui comparent X à l'un des éléments du tableau. Si X est plus petit, on le remplace par l'élément en question et, s'il est plus grand ou égal, on passe à la valeur suivante.

CMPD CMPS CMPU CMPY

Ces quatre instructions servent à établir une comparaison entre le registre spécifié et un nombre de 16 bits. Modes d'adressage : immédiat, indexé et étendu.

LEAX

Abréviation de Load Effective Adress, cette instruction charge le registre X avec une nouvelle adresse. On ne peut utiliser que le mode d'adressage indexé.

Exemple : MODE D'ADRESSAGE INDEXÉ (26 octets)

1. Programme BASIC

```
10
20
30 EXEC 32701
```

2. Programme assembleur

Lignes	Codes machine	Assembleur		
1	8E 40 00		LDX	#16384
2	86 FF		LDA	#255
3	A7 84	SUITE1	STA	,X
4	30 88 51		LEAX 81	,X
5	8C 4C 57		CMPX	#19543
6	26 F6		BNE	SUITE1(−10)
7	A7 84	SUITE2	STA	,X
8	30 88 4F		LEAX 79	,X
9	8C 58 AF		CMPX	#22703
10	26 F6		BNE	SUITE2(−10)
11	39		RTS	

Lignes 1 et 2 : X contient l'adresse du premier segment écran et A va permettre d'allumer les huit points des différents segments.

Ligne 3 : la valeur 255 est écrite dans l'octet 16384, et le segment situé en haut et à gauche de l'écran s'allume.

Ligne 4 : voici notre nouvelle instruction : LEAX va écrire un nouveau nombre de 16 bits dans le registre X. Le mode d'adressage indexé indique que X sera chargé avec l'ancienne valeur de ce registre à laquelle 81 sera ajouté. L'adresse effective contenue dans X est passée de 16384 à 16465 et le point de l'écran concerné sera donc deux lignes en dessous du précédent (2 fois 40) et une case à droite (+1).

Lignes 5 et 6 : on regarde si la valeur écrite dans le registre est égale ou non à 19543. La réponse négative entraîne le programme à la ligne SUITE1 ; 255 est écrit dans l'octet 16465 et le deuxième segment s'allume. On détermine alors l'adresse du troisième segment en ajoutant 81 à X. Une nouvelle comparaison est ensuite faite avec 19543 et le retour à la ligne 3 est à nouveau effectué. Le troisième segment est lui aussi décalé de deux lignes vers le bas et d'une case vers la droite par rapport au précédent.

Puisque 19543 = 16384 + 39*81, on aura au total 39 segments dessinés sur l'écran et ceci formera un escalier descendant sur la droite.

Ligne 7 : il nous manquait la quarantième marche : la voilà.

Lignes 8, 9 et 10 : on retrouve la même boucle que précédemment mais cette fois-ci, c'est 79 qui est ajouté à X. Faisons les comptes : les segments qui vont s'allumer maintenant seront eux aussi placés deux lignes en dessous (2 fois 40), mais dans la case de gauche (−1). On dessine donc un deuxième escalier qui partira de la droite de l'écran et qui descendra vers la gauche. Le nombre 22703, mais vous vous en doutiez, correspond à 19543 + 40*79.

LEAU LEAS LEAY

Ces trois instructions permettent de placer une nouvelle valeur dans les registres U, S et Y. Il n'y a que le mode d'adressage indexé qui est utilisable.

ORA

Un OU logique est effectué entre l'accumulateur A et le contenu d'un octet ou entre l'accumulateur et un nombre de 8 bits. Les modes d'adressage permis sont l'immédiat, l'indexé et l'étendu.

Exemple : MODE D'ADRESSAGE IMMÉDIAT (20 octets)

1. Programme BASIC

```
10
20
30 INPUT "DONNEZ-MOI UN NOMBRE" ; N
40 POKE 32750 , N : EXEC 32701
50 ON PEEK (32751) GOTO 60, 70
60 PRINT "LE NOMBRE EST IMPAIR" : GOTO 30
70 PRINT "LE NOMBRE EST PAIR" : GOTO 30
```

2. Programme assembleur

Lignes	Codes machine	Assembleur		
1	8E 7F EE		LDX	#32750
2	A6 84		LDA	,X
3	8A 01		ORA	#1
4	A1 84		CMPA	,X
5	26 04		BNE	PAIR(+4)
6	86 01		LDA	#1
7	20 02		BRA	FIN (+2)
8	86 02	PAIR	LDA	#2
9	A7 01	FIN	STA	1, X
10	39		RTS	

Réservons quelques lignes pour revoir de quelle façon s'exécute un OU logique entre deux nombres :

```
      1100
   OU 1010
   -------
   =  1110
```

Notre programme, pour sa part, va effectuer un OU entre le contenu du registre A et le nombre 1. Puisque 1 s'écrit en binaire 00000001, seul le dernier bit de A sera concerné. Ainsi, si A se termine par 0, il se verra modifié car son dernier chiffre passera à 1. Par contre, si son dernier chiffre vaut 1, A gardera la même valeur.

Lignes 1 et 2 : le nombre que l'on a proposé à l'ordinateur a été rentré par POKE dans l'octet 32750 et c'est donc le registre A qui est chargé avec cette valeur.

Ligne 3 : le OU logique est réalisé entre le nombre que nous avons choisi et l'unité. Si ce nombre est pair, son écriture binaire se fera avec un 0 à la fin et, s'il est impair, il se terminera par 1. L'action de ORA va donc consister à modifier la valeur de notre registre uniquement dans le cas où il est pair.

Lignes 4 et 5 : comparaison est faite entre les contenus de l'accumulateur et de l'octet 32750, octet qui, ne l'oublions pas, contient le nombre que nous avons tapé au clavier. Dans le cas où ce nombre est pair, ORA l'a transformé et un branchement à la ligne 8 est effectué.

Lignes 6 et 7 : s'il s'agit d'un nombre impair, la valeur 1 est mise dans A pour être réécrite ensuite (ligne 9) dans l'octet 32751.

Ligne 8 : sinon, c'est le nombre 2 qui va alors transiter par l'accumulateur pour être placé ensuite dans ce même octet.

La ligne BASIC 50 va alors examiner cet octet et le branchement ON GOTO enverra à ce moment l'ordinateur à la bonne instruction.

ORB

Un OU logique est réalisé entre le registre B et une valeur 8 bits. On peut employer les modes d'adressage immédiat, indexé et étendu.

ANDA

ANDA réalise un ET logique entre l'accumulateur et une donnée de 8 bits. On peut utiliser les modes d'adressage immédiat, indexé et étendu.

Exemple : MODE D'ADRESSAGE IMMÉDIAT (31 octets)

1. Programme BASIC

```
 5 CLS
10
20
30 PRINT "CHOISISSEZ LA COULEUR DE LA LIGNE"
40 INPUT "DE 0 A 7" ; N
50 POKE 32750 , 8*N : EXEC 32701 : GOTO 30
```

2. Programme assembleur

Lignes	Codes machine	Assembleur		
1	C6 FF		LDB	#255
2	8E 4F A0		LDX	#20384
3	B6 E7 C3	SUITE	LDA	> 59331
4	8A 01		ORA	#1
5	B7 E7 C3		STA	> 59331
6	E7 B4		STB	,X
7	84 FE		ANDA	#254
8	B7 E7 C3		STA	> 59331
9	B6 7F EE		LDA	> 32750
10	A7 80		STA	, X+
11	8C 4F C8		CMPX	#20424
12	26 E7		BNE	SUITE(–25)
13	39		RTS	

Avec cette nouvelle instruction, nous allons pouvoir choisir la couleur de nos dessins. Le lecteur pourra, dès cette étude terminée, essayer de reprendre le programme qui a été rencontré dans le Chapitre 2. Beaucoup de choses doivent maintenant être claires dans son esprit. Pour l'instant, reprenons ce qui avait été dit à propos du bit 0 de l'octet 59331 :

— si ce bit est à 1, on choisit la forme du segment.

— s'il est à 0, on peut en déterminer la couleur.

Lignes 1 et 2 : c'est l'initialisation des registres B et X. B va nous servir à allumer les huit points des segments et X contient l'adresse d'un segment situé à gauche de l'écran et en son milieu.

Lignes 3, 4 et 5 : le contenu de l'octet 59331 est placé dans A, un OU logique est effectué entre le registre et l'unité ; et le nouveau contenu de A est remis dans l'octet. Nous sommes sûrs, à ce moment-là, que le bit 0 de l'octet en question est à 1.

Ligne 6 : les huit points du segment 20384 s'allument avec la couleur qui était en service dans le BASIC.

Lignes 7 et 8 : un ET logique est réalisé entre l'accumulateur et le nombre 11111110 en binaire. Ceci a pour effet de ne modifier aucun des sept premiers chiffres de l'octet 59331 tout en obligeant le dernier chiffre à valoir 0. On va donc pouvoir décider de la couleur du segment.

Lignes 9 et 10 : on retrouve dans l'octet 32750 la couleur qui y a été placée par POKE et on colorie le premier segment. En même temps s'effectue une incrémentation du registre X qui va donc contenir l'adresse d'un segment placé exactement à droite du précédent.

Lignes 11 et 12 : on repart dans la boucle une fois que la comparaison de X et du nombre 20424 a fait apparaître une différence.

Voici donc une boucle qui va s'exécuter 40 fois, allumant de ce fait les 40 segments qui constituent la ligne horizontale visible au milieu de l'écran.

Pour finir, une remarque sur le dernier bit de l'octet 59331 : il est généralement positionné à 1 par l'ordinateur mais attention, ceci n'est pas toujours vrai ! Il vaut par exemple 0 après l'exécution de l'instruction CLS : pensez alors à le ramener à 1 si votre programme assembleur doit dessiner des segments sur l'écran.

ANDB

Le registre B et un nombre 8 bits font l'objet d'un ET logique. Les modes d'adressage immédiat, étendu et indexé peuvent être utilisés.

EORA

Comme les deux instructions précédemment étudiées, EORA réalise une opération logique : un OU exclusif est effectué entre le registre A et un nombre de huit bits. On peut là aussi, utiliser les modes d'adressage immédiat, indexé et étendu.

Exemple : MODE D'ADRESSAGE IMMÉDIAT (11 octets)

1. Programme BASIC

```
 5 REMABCDEFGHIJKLM
10 FOR I = 26106 TO 26116 : READ I$ : POKE I, VAL("&H" + I$) :
   NEXT
20 DATA B6, 66, 05, 88, FF, 8B, 01, B7, 66, 06, 39
30 PRINT "QUEL NOMBRE NEGATIF VOULEZ-VOUS ECRIRE"
40 INPUT "EN COMPLEMENT A DEUX" ; N
50 POKE 26117 , -N : EXEC 26106
60 PRINT "LA REPONSE VAUT" ; PEEK(26118)
70 PRINT "SOIT" ; HEX$(PEEK(26118)) ; "EN HEXADECIMAL"
```

2. Programme assembleur

Lignes	Codes machine	Assembleur
1	B6 66 05	LDA >26117
2	88 FF	EORA #255
3	8B 01	ADDA #1
4	B7 66 06	STA >26118
5	39	RTS

L'utilisation du OU exclusif est faite ici dans le but d'inverser tous les chiffres d'un nombre binaire.

```
      1           0
  EOR 1       EOR 1
  -----       -----
  =   0       =   1
```

Quand un OU exclusif est effectué entre un chiffre et 1, ce chiffre change de valeur.

Ligne 1 : on récupère dans l'octet 26117 le nombre que POKE y a placé durant l'exécution du programme BASIC.

Ligne 2 : on réalise un OU exclusif entre ce nombre et 255. Comme 255 s'écrit 11111111 en binaire, tous les chiffres de notre nombre seront inversés.

Ligne 3 : pour avoir le complément à deux, il ne reste qu'une opération à faire : l'addition d'une unité.

Ligne 4 : le résultat est rangé dans l'octet 26118 et les deux dernières lignes du programme BASIC n'ont plus qu'à nous donner la réponse.

Pas de difficulté donc pour interpréter le programme assembleur. Profitons-en pour regarder de plus près la façon dont les codes machine ont, cette fois, été chargés en mémoire centrale.

Le BASIC est rangé dans la mémoire à partir de l'octet 26101 et voici comment :

```
FOR I = 26101 TO 26119 : PRINT PEEK (I) ; : NEXT
```

La réponse est :

```
102  8  0  5  140  65  66  67 ... 77  0
```

102 et 8 donnent avec la méthode poids forts/poids faibles le nombre 26120 ;

0 et 5 donnent, de même, 5

140 est l'un des nombres réservés aux mots clés : c'est le code de REM

65 est le code ASCII de A

66 est le code ASCII de B

77 est celui de M

0 est le chiffre qui indique que la ligne BASIC est terminée.

Toutes ces valeurs correspondent à la première ligne :

```
5 REMABCDEFGHIJKLM
```

26120 est l'adresse du premier octet de la ligne 10.

5 est le numéro de la ligne

140, 65, 66 ... 77 sont les nombres correspondant à ce qui a été tapé au clavier.

Et notre sous-programme assembleur dans tout cela ? Il a été logé dans la mémoire à partir de l'octet 26106 : le BASIC ne va donc pas avoir à l'interpréter puisque, dès qu'il rencontre une REM, il passe directement à la ligne suivante. Voici donc définie une nouvelle façon d'implanter les codes machines en mémoire. Naturellement, il faudra toujours prévoir un nombre suffisant d'octets en tapant la série de caractères quelconques après le mot REM.

EORB

L'opération logique OU exclusif est effectuée entre le registre B et un nombre de 8 bits. Sont permis les modes d'adressage immédiat, étendu et indexé.

PSHU PULU

PSHU est l'abréviation de PuSH registers in to User stack.

PULU est celle de PULl registers from User stack.

Ces deux instructions permettent l'empilement et le dépilement de registres dans la pile utilisateur. Elles n'autorisent que le mode d'adressage immédiat.

Exemple : MODE D'ADRESSAGE IMMÉDIAT (24 octets)

1. Programme BASIC

```
10
20
30 EXEC 32701
```

2. Programme assembleur

Lignes	Codes machine	Assembleur		
1	CE 7F FF		LDU	#32767
2	8E 4F A0		LDX	#20384
3	86 FF		LDA	#255
4	C6 14		LDB	#20
5	36 02	SUITE	PSHU	A
6	86 10		LDA	#16
7	A7 80		STA	,X+
8	37 02		PULU	A
9	A7 80		STA	,X+
10	5A		DECB	
11	26 F3		BNE	SUITE(−13)
12	39		RTS	

Ligne 1 : on détermine à partir de quel endroit de la mémoire doit se placer la pile. Ici c'est le dernier octet de la mémoire vive qui a été choisi.

Lignes 2, 3 et 4 : 20384 est l'adresse d'un segment qui se trouve au milieu de l'écran et sur la droite. A contient pour l'instant la valeur 255 et B, qui nous servira de compteur, est chargé avec le nombre 20.

Ligne 5 : on empile la valeur de l'accumulateur et U passe à 32766 — n'oublions pas que la pile va vers le bas dans le TO 7. Ce qui fait que l'ordinateur a mis dans le premier octet de la pile (le numéro 32766) la valeur du registre A.

Lignes 6 et 7 : A est chargé avec un nouveau nombre, 16, et ce nombre est transféré dans l'octet pointé par X. Puisque 16 s'écrit 00010000 en binaire, un seul point du segment 20384 s'allume. Puis X est incrémenté.

Ligne 8 : on va chercher dans la pile le nombre qui y était, 255 en l'occurrence, et on l'écrit dans A.

Ligne 9 : 255, la nouvelle valeur de A, est placé dans l'octet 20385 et tous les points du segment correspondant s'allument. Pour l'instant, sur l'écran, apparaît donc un point suivi d'un segment.

Lignes 10 et 11 : le deuxième accumulateur est décrémenté et vaut de ce fait 19. L'instruction BNE renvoie alors le programme à la ligne 5. PSHU remet de côté le nombre 255 et la boucle est lue une deuxième fois. 255 sera ensuite retrouvé, grâce à la pile, à la ligne 8.

En définitive, après vingt passages, une ligne alternant points et traits sera dessinée sur le téléviseur.

Voici donc illustré l'intérêt de la pile : elle permet de gérer la pénurie liée au très petit nombre de registres d'un microprocesseur.

Pour terminer, il nous faut voir de quelle manière s'obtiennent les codes machine qui suivent les instructions PULU et PSHU.

128	64	32	16	8	4	2	1
PC	S	Y	X		B	A	

——→ empilement dépilement ←——

PSHU A est codée 36 02

PSHU B est codée 36 04

PSHU X est codée 36 10 (10 est la valeur hexa de 16)

PSHU B S est codée 36 44 (44 est la valeur hexa de 68)

Cette dernière instruction sert à empiler d'abord S et ensuite B.

PSHS PULS

Ces deux instructions servent à empiler ou dépiler une combinaison de registres de la pile système S. On ne peut employer que le mode d'adressage immédiat.

Exemple : MODE D'ADRESSAGE IMMÉDIAT (38 octets)

1. Programme BASIC

```
10
20
30 PRINT "CHOISISSEZ LA COULEUR DE L'ECRAN"
40 INPUT "DE 0 A 7" ; N
50 POKE 32750 , N:EXEC 32701
```

2. Programme assembleur

Lignes	Codes machine	Assembleur		
1	8E 40 00		LDX	#16384
2	B6 7F EE		LDA	> 32750
3	34 02	SUITE	PSHS	A
4	B6 E7 C3		LDA	> 59331
5	8A 01		ORA	#1
6	B7 E7 C3		STA	> 59331
7	34 02		PSHS	A
8	4F		CLRA	
9	A7 84		STA	,X
10	35 02		PULS	A
11	84 FE		ANDA	#254
12	B7 E7 C3		STA	> 59331
13	35 02		PULS	A
14	A7 80		STA	,X+
15	8C 5F 40		CMPX	#24384
16	25 E1		BLO	SUITE(−31)
17	39		RTS	

Ce programme reconstitue la fonction CLS, avec la possibilité de choisir la couleur.

Lignes 1, 2 et 3 : X contient l'adresse du premier octet de l'écran et A est chargé avec le nombre correspondant à la couleur que nous avons choisie. Cette couleur est sauvegardée dans la pile système au niveau de la ligne 3.

Lignes 4, 5 et 6 : on force à 1 le bit de droite de l'octet 59331 pour pouvoir choisir la configuration du premier segment écran.

Ligne 7 : l'accumulateur est placé dans la pile, ce qui revient à mettre de côté la valeur de l'octet 59331.

Lignes 8 et 9 : 0 est écrit dans l'octet 16384 et, de ce fait, les huit points du premier segment sont de la couleur du fond.

Lignes 10, 11 et 12 : on retrouve l'octet 59331, et on passe son dernier bit à 0. On va pouvoir choisir les couleurs.

Ligne 13 : on extrait de la pile un nombre pour l'écrire dans le registre A. Cette fois, l'instruction PULS correspond au PSHS de la ligne 3 et c'est donc le code de la couleur qui est placé dans A.

Ligne 14 : le segment se colorie maintenant de la façon que nous avions souhaitée. Théoriquement, puisque nous devions choisir un nombre entre 0 et 7, il devrait s'allumer en noir sur fond de couleur. Mais, bien entendu, seuls les points de fond sont concernés dans cet exemple et le noir n'apparaît pas.

Lignes 15 et 16 : le registre X a été incrémenté à la ligne précédente et vaut donc 16385. L'instruction BLO va alors renvoyer l'ordinateur à SUITE, 31 octets en arrière.

On a ici affaire à une boucle qui va être exécutée 8000 fois et, à son terme, l'écran aura pris la couleur désirée.

Le tableau permettant de coder les registres à empiler ou à dépiler est presque le même que celui rencontré avec l'autre pile. Seul le bit 6 est différent : il concerne cette fois le registre U.

LSRA

Abréviation de Logical Shift Right, *cette instruction décale tous les bits de l'accumulateur A vers la droite. Le bit de gauche est mis à 0. On ne peut utiliser que le mode d'adressage inhérent.*

Exemple : MODE D'ADRESSAGE INHÉRENT (8 octets)

1. Programme BASIC

```
10
20
30 PRINT "DONNEZ UN NOMBRE PLUS PETIT QUE 256" ;
40 INPUT N : POKE 32750 , N : EXEC 32701
50 PRINT "LE QUOTIENT ENTIER DU NOMBRE" ;
60 PRINT "PAR DEUX VAUT" ; PEEK (32751) : GOTO 30
```

2. Programme assembleur

Lignes	Codes machine	Assembleur	
1	B6 7F EE	LDA	> 32750
2	44	LSRA	
3	B7 7F EF	STA	> 32751
4	39	RTS	

Ce programme effectue en assembleur la division entière d'un nombre par deux. Voyons au niveau du binaire, comment cela se passe.

Considérons le nombre décimal 100 qui s'écrit en binaire 01100100.

REGISTRE A

0 → | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | → C

bit 7 bit 0

Faisons subir aux chiffres qui constituent ce nombre un décalage sur la droite. Chaque chiffre va se retrouver dans le bit de rang immé-

diatement inférieur : le chiffre du bit 7, 0, va passer dans le bit 6, le chiffre 1 du bit 6 va s'écrire dans le bit 5, etc. Le dernier chiffre à droite (bit 0) va donc sortir de l'octet et sera perdu pour nous. L'ordinateur, lui, en gardera la trace en le mettant dans un endroit spécial qu'on appelle l'indicateur de retenue et que l'on note C. Cet indicateur prendra donc la valeur 0, mais, répétons-le, ceci n'a aucune espèce d'importance pour notre exemple.

Sachant que LSRA remplace toujours le bit 7 par 0, voici ce que l'on obtient alors pour le registre A :

0	0	1	1	0	0	1	0

La traduction en décimal de cette valeur donne 50 ; on a donc bien divisé le nombre 100 par 2.

Et si nous étions partis d'un nombre impair ? Essayons avec 101.

0 → | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | ← C

Quand LSRA aura agit, on obtiendra :

0	0	1	1	0	0	1	0

c'est-à-dire 50, ce qui est bien le quotient entier de 101 par 2. Dans ce deuxième cas, l'indicateur de retenue est passé à 1.

Il ne reste plus qu'à comprendre pourquoi le décalage à droite des chiffres a conduit à une division par deux. Prenons par exemple le chiffre 1 du bit 6 et voyons ce qu'il devient : il valait 2^6, c'est-à-dire 64 avant LSRA, il vaut 2^5, soit 32, après ; il a donc été réduit de moitié. Ce même raisonnement se fait pour tous les autres chiffres, ce qui nous donne l'explication.

Les différentes lignes ne seront pas étudiées une à une cette fois-ci, le programme assembleur se comprenant sans difficulté.

LSRB

Un décalage de tous les bits du registre B est effectué sur la droite. Le bit 7 s'annule et le bit 0 est placé dans l'indicateur de retenue. C'est le mode d'adressage inhérent qui est employé.

LSR ,X

Comme pour LSRA, c'est un décalage sur la droite, mais ce décalage concernera le contenu d'un octet mémoire. Les modes d'adressage indexé et étendu sont permis.

Exemple : MODE D'ADRESSAGE INDEXÉ (30 octets)

1. Programme BASIC

```
10
20
30 EXEC 32701
40 FOR I = 1 TO 7 : PLAY "PP" : EXEC 32717 : NEXT
```

2. Programme assembleur

Lignes	Codes machine	Assembleur		
1	8E 40 00		LDX	#16384
2	86 80		LDA	#128
3	A7 84	SUITE1	STA	,X
4	30 88 28		LEAX 40	,X
5	8C 5F 40		CMPX	#24384
6	25 F6		BLO	SUITE1(−10)
7	39		RTS	
8	8E 40 00		LDX	#16384
9	64 84	SUITE2	LSR	,X
10	30 88 28		LEAX 40	,X
11	8C 5F 40		CMPX	#24384
12	25 F6		BLO	SUITE2(−10)
13	39		RTS	

Lignes 1 et 2 : c'est l'initialisation du programme. On partira du premier segment de l'écran et A va contenir le nombre qui s'écrit 10000000 en binaire.

Lignes 3, 4, 5 et 6 : A est écrit dans l'octet 16384, ce qui allume le premier point du segment correspondant. On ajoute alors 40 à X qui va pointer sur le segment situé exactement au-dessous du précédent. On vérifie que l'on ne sort pas de la plage écran, et on retourne à SUITE1. Cette fois c'est le premier point du segment 16424 qui s'allume et X se voit à nouveau ajouter le nombre 40. Après une comparaison avec 24384, l'ordinateur remonte à la ligne 3.

Il s'agit en conclusion d'une boucle qui sera exécutée 200 fois, et la première colonne de l'écran sera entièrement allumée. L'instruction RTS de la ligne 7 fait que le BASIC retrouve son exécution normale et qu'il nous conduit à la boucle FOR NEXT.

Cette boucle laisse passer un peu de temps et renvoie le microprocesseur dans le programme machine. Mais attention, cette fois-ci l'exécution a lieu à partir de l'octet 32717, c'est-à-dire à la ligne 8 de l'assembleur.

Lignes 8 et 9 : on repart du début de la mémoire écran et l'octet 16384 subit un décalage sur la droite. Puisqu'il valait en binaire 10000000, il vaudra, toujours en binaire, 01000000 et c'est donc le deuxième point du premier segment de l'écran qui va s'allumer.

Lignes 10, 11 et 12 : on charge le registre X avec l'adresse du segment placé au-dessous du précédent, et on regarde si l'on n'atteint pas le bas de l'image. On retrouve alors une boucle qui tracera la deuxième colonne de l'écran, la première s'étant effacée.

On ressort à ce moment-là du programme assembleur, une temporisation est effectuée par l'instruction PLAY "PP" et on se replace à nouveau à la ligne 8. Pour y tracer cette fois une troisième colonne, colonne décalée d'un point sur la droite par rapport à la deuxième.

La ligne BASIC FOR NEXT est exécutée sept fois, ce qui fait qu'au total huit colonnes seront apparues les unes à la suite des autres sur le téléviseur.

LSLA

C'est cette fois-ci d'un décalage vers la gauche qu'il est question, LSL étant l'abréviation de Logical Shift Left. Le bit 7 passe dans l'indicateur de retenue et le bit 0 du registre A est mis à zéro. LSLA est utilisée avec le seul mode d'adressage inhérent.

Exemple : MODE D'ADRESSAGE INHÉRENT (8 octets)

1. Programme BASIC

```
10
20
30 INPUT "DONNEZ-MOI UN NOMBRE" ; N : POKE 32750 , N
40 J = 1 : FOR I = 1 TO 3
50 J = J*2 : EXEC 32701
60 PRINT "LE PRODUIT DU NOMBRE PAR" ; J ;
70 PRINT "VAUT :" ; PEEK (32750)
80 NEXT : GOTO 30
```

2. Programme assembleur

Lignes	Codes machine	Assembleur
1	B6 7F EE	LDA > 32750
2	48	LSLA
3	B7 7F EE	STA > 32750
4	39	RTS

On suppose que l'on propose à l'ordinateur le nombre 31 et on regarde ce qu'il devient quand on exécute l'instruction LSLA. 31 a pour équivalent binaire 00011111.

REGISTRE A

C ← | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | ← 0

bit 7 bit 0

Le décalage à gauche va faire sortir du registre le contenu du bit 7 qui ira se placer dans l'indicateur de retenue, indicateur dont l'importance est nulle en l'état d'avancement de nos connaissances. Tous les chiffres étant translatés, on obtient pour A :

0	0	1	1	1	1	1	0

Le nombre 0 est venu prendre la place laissée libre dans le bit 0. En transcrivant le résultat obtenu en décimal, on obtient le nombre 62 c'est-à-dire le double de 31. Ainsi le décalage à gauche de tous les bits a permis d'effectuer le produit par 2 du nombre qui se trouvait dans l'accumulateur. Ceci peut se comprendre puisque, en définitive, chaque chiffre se retrouvera avec une puissance de 2 supérieure d'une unité à la précédente.

Revenons à notre programme : le nombre que l'on a donné au départ à l'ordinateur est placé dans l'octet 32750 et, au premier passage de la boucle FOR NEXT, ce nombre est multiplié par 2. La ligne assembleur 3 replace la réponse dans ce même octet 32750. Au deuxième passage, le nombre est à nouveau multiplié par 2, ce qui fait que la valeur de début est, cette fois, multipliée par 4 ; elle le sera par 8 quand le programme BASIC sera terminé.

Bien entendu, ce programme donne des réponses cohérentes tant que l'on ne choisit pas un nombre supérieur ou égal à 32 (soit 00100000 en binaire). A partir de cette valeur, en effet, c'est un chiffre 1 qui est perdu dans les décalages rendant le résultat final incorrect (mais explicable).

LSLB

Tous les bits du registre B sont décalés sur la gauche. Le bit de droite s'annule et celui de gauche passe dans l'indicateur de retenue. On emploie le mode d'adressage inhérent seulement.

LSL

Un décalage d'un bit sur la gauche du contenu d'un octet mémoire est effectué. Le bit 0 passe à 0 et le bit 7 est écrit dans l'indicateur de retenue. Cette instruction s'exécute avec les modes d'adressage indexé et étendu.

Exemple : MODE D'ADRESSAGE INDEXÉ (30 octets)

1. Programme BASIC

```
10
20
30 X = 24383
40 POKE 32750, INT (X/256)
50 POKE 32751, X - INT (X/256)*256
60 EXEC 32701 : X = X - 41 : IF X > = 22784 THEN 40
```

2. Programme assembleur

Lignes	Codes machine	Assembleur		
1	86 01		LDA	#1
2	C6 08		LDB	#8
3	BE 7F EE		LDX	> 32750
4	A7 84		STA	,X
5	68 84	SUITE1	LSL	,X
6	34 04		PSHS	B
7	8D 06		BSR	TEMPO(+6)
8	35 04		PULS	B
9	5A		DECB	
10	26 F5		BNE	SUITE1(−11)
11	39		RTS	
12	CC 27 10	TEMPO	LDD	#10000
13	83 00 01	SUITE2	SUBD	#1
14	26 FB		BNE	SUITE2(−5)
15	39		RTS	

Lignes 1, 2, 3 et 4 : les registres A, B et X sont chargés respectivement par les nombres 1, 8 et 24383. Ce dernier est l'adresse du segment placé en bas et à droite de l'écran. L'exécution de l'instruction STA ,X allume le point de droite du segment en question.

Ligne 5 : c'est le début d'une boucle qui sera utilisée huit fois. Un décalage des bits de l'octet 24383 est réalisé et cet octet prend alors la valeur binaire 00000010 : c'est donc le deuxième point en partant de la droite qui va cette fois s'éclairer.

Ligne 6 : on sauvegarde dans la pile système la valeur du registre B car celui-ci va être utilisé dans la boucle d'attente.

Ligne 7 : un branchement vers le sous-programme de temporisation (ligne 12) est effectué. On fait perdre à ce moment-là un peu de temps au microprocesseur en l'obligeant à compter jusqu'à 10000, puis l'instruction RTS nous ramène à la ligne 8. Ainsi les commandes BSR et RTS des lignes 7 et 15 auront fonctionné exactement de la même façon que l'aurait fait le couple BASIC GOSUB-RETURN.

Lignes 8, 9 et 10 : on réécrit dans le registre B la valeur qu'il possédait avant la temporisation, et on le décrémente : il passe alors à 7. Puisqu'il n'est pas nul, l'ordinateur est rebranché à la ligne SUITE1 pour une deuxième exécution de la boucle principale. Les bits de l'octet 24383 sont à nouveau décalés et c'est le troisième point du segment correspondant qui s'allume.

Quand l'instruction RTS de la ligne 11 sera réalisée, les huit points du dernier segment de l'écran auront été rendus visibles puis effacés les uns après les autres.

Retour au BASIC : on retranche 41 au nombre 24383 et c'est le segment situé à gauche et au-dessus du précédent qui va être concerné ; l'adresse de ce segment est communiquée au système par les instructions POKE des lignes 40 et 50.

On replonge ensuite dans la partie assembleur et les huit points du segment 24342 sont allumés les uns à la suite des autres et de droite à gauche.

Voici donc expliqué ce que nous pouvons voir sur notre écran, un point se déplaçant sur la gauche et remontant d'une position à intervalles réguliers. Le nombre 22784 du programme BASIC est égal à 24383 – 39*41, mais vous l'auriez deviné, non ?

ROLA

Tous les bits de l'accumulateur subissent une rotation vers la gauche. Le bit 7 passe dans l'indicateur de retenue et la valeur préalablement contenue par celui-ci est transférée dans le bit 0. ROLA est l'abréviation de ROtate Left A. *Seul le mode d'adressage inhérent est autorisé.*

Exemple : MODE D'ADRESSAGE INHÉRENT (15 octets)

1. Programme BASIC

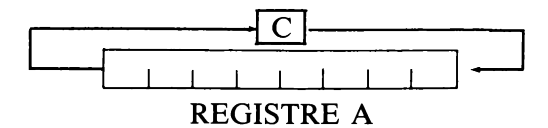
```
10
20
30 INPUT "DONNEZ UN NOMBRE" ; N : POKE 32750 , N
40 EXEC 32701 : PRINT "SON DOUBLE VAUT :" ;
50 PRINT 256*PEEK (32751) + PEEK (32752) : GOTO 30
```

2. Programme assembleur

Lignes	Codes machine	Assembleur	
1	4F	CLRA	
2	5F	CLRB	
3	49	ROLA	
4	B6 7F EE	LDA	> 32750
5	49	ROLA	
6	59	ROLB	
7	F7 7F EF	STB	> 32751
8	B7 7F F0	STA	> 32752
9	39	RTS	

Vous vous souvenez de l'instruction LSLA ? Elle nous avait permis de multiplier un nombre par 2, 4 ou 8 mais cela n'était pas allé

sans un ennui de taille : les nombres 1 qui sortaient sur la gauche de l'accumulateur étaient perdus et, si l'on partait d'un nombre trop grand, la réponse n'était pas celle attendue. Regardons comment nous allons pouvoir y remédier avec notre nouvelle instruction ROLA :

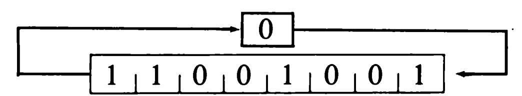


Tous les bits de l'accumulateur subissent un décalage sur la gauche ; le bit contenu dans l'indicateur de retenue passe dans le bit 0 et c'est le bit 7 qui prend sa place. Il s'agit donc là d'une rotation réalisée sur 9 bits.

Lignes 1 et 2 : les deux registres 8 bits sont mis à zéro.

Ligne 3 : on fait subir à A une rotation ; puisque A s'écrit 00000000 en binaire, ceci n'a pas d'autre effet que de faire rentrer le chiffre 0 dans l'indicateur de retenue.

Lignes 4 et 5 : on recopie dans A le nombre que nous avons écrit par POKE dans l'octet 32750 et, grâce à ROLA, on le multiplie par 2. Examinons cela de plus près et supposons, pour fixer les idées, que N ait été choisi égal à 201 (soit 11001001).



C est à 0 (ligne 3) et on obtient donc après ROLA :

1

1	0	0	1	0	0	1	0

Le bit C est passé à 1 et la nouvelle valeur de l'accumulateur est, en décimal, 146. Ceci n'est naturellement pas le double de 201, mais attendons la suite.

Lignes 6 et 7 : ROLB a pour effet de décaler les 8 chiffres 0 du registre B et de faire entrer sur sa droite le bit qui se trouvait dans l'indicateur, c'est-à-dire le bit 1. La nouvelle valeur de B est donc 1 ; elle est inscrite alors dans l'octet 32751.

Lignes 8 et 9 : le décimal 146 est, pour sa part, placé à l'adresse 32752 et le retour au BASIC est programmé.

On va pouvoir vérifier la logique du programme assembleur :

```
PRINT 256*PEEK (32751) + PEEK (32752)
```

Réponse : 256*1 + 146 = 402.

Tout s'est donc passé en définitive comme si nous avions fait un décalage sur 9 bits.

011001001 (201 décimal) serait devenu 110010010 (402 décimal).

ROLB

Cette instruction agit sur B de la même façon que ROLA le fait sur A. C'est le mode d'adressage inhérent qui doit être utilisé.

ROL

Tous les bits de l'octet mémoire spécifié sont décalés d'une position sur la gauche. Le bit 7 est placé dans l'indicateur de retenue et la valeur d'origine de celui-ci est transférée dans le bit 0. Les modes d'adressage possibles sont l'indexé et l'étendu.

Exemple : MODE D'ADRESSAGE INDEXÉ (31 octets)

1. Programme BASIC

```
 5 CLS
10
20
30 EXEC 32701
```

2. Programme assembleur

Lignes	Codes machine	Assembleur		
1	B6 E7 C3		LDA	>59331
2	8A 01		ORA	#1
3	B7 E7 C3		STA	>59331
4	8E 40 00		LDX	#16384
5	86 C8		LDA	#200
6	34 02	COL	PSHS	A
7	C6 28		LDB	#40
8	86 80	LIGNE	LDA	#128
9	49		ROLA	
10	69 80		ROL	,X+
11	5A		DECB	
12	26 F8		BNE	LIGNE(−8)
13	35 02		PULS	A
14	4A		DECA	
15	26 EF		BNE	COL(−17)
16	39		RTS	

Lignes 1, 2 et 3 : on force à 1 le bit 0 de l'octet 59331 puisque l'instruction CLS l'avait positionné à 0 et que nous voulons intervenir, non sur la couleur, mais sur la forme des segments. Le lecteur pourra court-circuiter ces trois lignes et lancer le programme machine avec EXEC 32709 ; il devra alors déterminer l'explication du phénomène qui apparaîtra sur son écran.

Lignes 4, 5 et 6 : X va contenir l'adresse du premier segment écran et A nous servira de compteur pour les 200 lignes de l'écran. On place dans la pile la valeur du registre car il va devoir être utilisé à une autre tâche.

Ligne 7 : voici le deuxième compteur, celui qui correspond à la largeur de l'écran de télévision.

Lignes 8 et 9 : on inscrit dans l'accumulateur la valeur décimale 128 et on lui fait subir une rotation sur la gauche à travers l'indicateur de retenue :

avant ROLA	1	0	0	0	0	0	0	0	?
après ROLA	0	0	0	0	0	0	0	?	1

La seule fonction de ces deux commandes est d'obliger le bit de retenue C à valoir 1. Notons que l'on ne peut prévoir à ce moment-là ce que contient le registre A.

Ligne 10 : on revient au premier segment de l'écran et l'on fait effectuer à ses bits une rotation sur la gauche. Du fait que toute l'image a été effacée par CLS, les octets écran ont tous une valeur nulle. Donc, en tenant compte de ce que l'indicateur de retenue est à 1, on obtient comme nouvelle valeur binaire de l'octet 16384 : 00000001. Ceci a, bien sûr, pour effet d'allumer le point de droite du premier segment de l'écran.

Lignes 11 et 12 : on recommence avec chacun des segments de la première ligne ce qui a été fait pour le premier.

Lignes 13, 14 et 15 : on retrouve le nombre 200, celui qui décompte les lignes, on le décrémente et on renvoie l'ordinateur à la ligne 6 pour qu'il s'occupe des 40 segments de la deuxième ligne de l'écran.

Quand l'exécution du programme sera terminée, les 8000 segments auront tous la même configuration : point de droite allumé. Une série de 40 lignes verticales sera dessinée sur le téléviseur.

RORA

Cette instruction effectue une rotation sur la droite de tous les bits de l'accumulateur A. Le bit de retenue prend la place du bit 7 ; il est lui-même remplacé par le bit 0. On utilise le mode d'adressage inhérent.

Exemple : MODE D'ADRESSAGE INHÉRENT (19 octets)

1. Programme BASIC

```
10
20
30 FOR I = 20384 TO 20423
40 POKE 32750, INT(I/256)
50 POKE 32751, I - INT(I/256)*256
60 EXEC 32701 : NEXT
```

2. Programme assembleur

Lignes	Codes machine	Assembleur		
1	BE 7F EE		LDX	> 32750
2	4F		CLRA	
3	46		RORA	
4	86 80		LDA	#128
5	C6 08		LDB	#8
6	A7 84	SUITE	STA	,X
7	30 88 D8		LEAX	−40,X
8	46		RORA	
9	5A		DECB	
10	26 F7		BNE	SUITE(−9)
11	39		RTS	

Ligne 1 : la valeur 20384, chargée par POKE dans les octets 32750 et 32751, est inscrite dans le registre X. 20384 est l'adresse d'un segment situé au milieu de l'écran et sur la gauche.

Lignes 2 et 3 : voici deux instructions dont la seule utilité consiste à forcer le bit de retenue, C, à 0.

Lignes 4 et 5 : le premier accumulateur est chargé par un nombre

qui vaut 10000000 en binaire, et le second par une valeur qui nous servira de compteur dans une boucle.

Lignes 6 et 7 : on écrit 128 dans le segment 20384 et son point de gauche s'allume. On retire alors 40 de X pour obtenir l'adresse du segment placé au-dessus du précédent (20344).

Ligne 8 : une rotation sur la droite du registre A est réalisée à travers l'indicateur de retenue. La valeur d'origine de A étant 128, on a :

	Retenue	Registre A
avant RORA	0	1 0 0 0 0 0 0 0
après RORA	0	0 1 0 0 0 0 0 0

Lignes 9 et 10 : on enlève une unité à B, on vérifie que ce registre n'est pas passé à 0 et on retrouve la ligne SUITE.

Pour, cette fois, ranger le contenu de l'accumulateur dans l'octet 20344. Le deuxième point, en partant de la gauche, du segment correspondant va s'allumer.

Au troisième passage de la boucle, c'est le troisième point du segment placé au-dessus du précédent qui va se voir. Et, quand le programme assembleur sera terminé, 8 points disposés en diagonale auront été éclairés.

Le BASIC, reprenant la conduite des opérations, calcule alors les poids fort et faible du nombre 20385 et les transmet à la machine pour que 8 nouveaux points soient allumés. Ceci va être répété 40 fois, l'ordinateur dessinant à chaque passage une petite diagonale.

Le lecteur pourra essayer de concevoir un programme équivalent à celui-ci, mais n'ayant pas l'obligation de retourner 40 fois au BASIC. En quelque sorte, un programme qui allume les 40 diagonales de façon autonome. Une méthode possible est d'empiler, au moment voulu, le registre X. Mais, attention, l'ordinateur utilise le bit de retenue lorsqu'il a affaire aux instructions suivantes : ADD, ADC, SUB, CLR, CMP, COM, NEG et MUL. Alors, évitez de les utiliser cette fois-ci !

RORB

Cette instruction permet de faire sur le registre B le même genre d'opération que RORA. On utilise le mode d'adressage inhérent.

ROR

Une rotation sur la droite du contenu d'un octet mémoire est réalisée. Le bit 0 prend la place du bit de retenue qui, lui-même, se retrouve à l'emplacement du bit 7. Il est permis d'utiliser les modes d'adressage indexé et étendu.

Exemple : MODE D'ADRESSAGE ÉTENDU (9 octets)

1. Programme BASIC

```
 10
 20
 30 PRINT "DONNEZ UN NOMBRE INFERIEUR A 65536"
 40 INPUT N : POKE 32750, INT(N/256)
 50 POKE 32751, N - INT(N/256)*256
 60 J = 1 : FOR I = 1 TO 4
 70 EXEC 32701 : J = J*2
 80 PRINT "LE QUOTIENT DU NOMBRE PAR" ; J ;
 90 PRINT "VAUT :" ; 256*PEEK (32750) + PEEK (32751)
100 NEXT : GOTO 30
```

2. Programme assembleur

Lignes	Codes machine	Assembleur
1	4F	CLRA
2	46	RORA
3	76 7F EE	ROR >32750
4	76 7F EF	ROR >32751
5	39	RTS

Ce programme exécute les divisions entières par 2, 4, 8 et 16 de n'importe quel nombre plus petit que 65536. Nous retrouvons donc une méthode déjà connue, le décalage sur la droite, mais nous allons l'appliquer ici, grâce à l'utilisation du bit de retenue, à une valeur écrite sur deux octets.

Prenons par exemple le nombre 1000 qui s'écrit en binaire 1111101000. Le programme BASIC le décompose en deux valeurs de 8 bits qui sont écrites dans l'octet 32750 pour le poids fort et dans l'octet 32751 pour le reste.

0	0	0	0	0	0	1	1

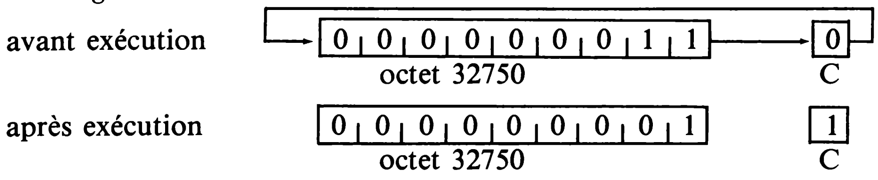
octet 32750

1	1	1	0	1	0	0	0

octet 32751

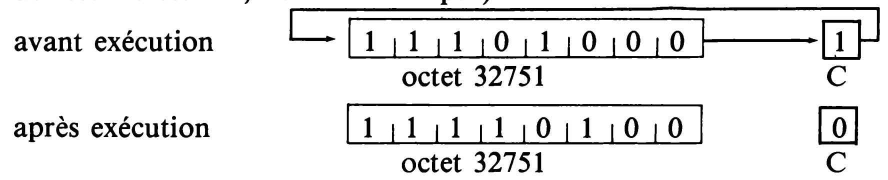
On procède maintenant de la même façon que la machine.

Ligne 3 : rotation des bits de l'octet 32750 sur la droite :



Vous aviez remarqué que la précaution avait été prise aux lignes 1 et 2 de placer le chiffre 0 dans l'indicateur de retenue.

Ligne 4 : rotation des bits de l'octet 32751 sur la droite (le bit de retenue est à 1, ne l'oublions pas) :



Faisons les comptes :
Les octets 32750 et 32751 valent respectivement en décimal 1 et 244. Lorsqu'on applique la règle poids fort/poids faible, on obtient :

1*256 + 244 = 500

Ceci est bien la réponse attendue.

Lorsqu'une nouvelle exécution du programme machine sera commandée par le BASIC, les octets 32750 et 32751 se verront décalés sur la droite, le bit sortant du premier étant réécrit au début du deuxième : c'est ainsi qu'une nouvelle division par 2 sera réalisée.

ADCA

Cette instruction est l'abréviation de ADd with Carry into A. Un nombre de 8 bits, le contenu de l'indicateur de retenue et la valeur de A sont ajoutés. Le résultat est mis dans l'accumulateur. Les modes immédiat, indexé et étendu sont autorisés.

Exemple : MODE D'ADRESSAGE IMMÉDIAT (17 octets)

1. Programme BASIC

```
 10
 20
 30 INPUT "PREMIER NOMBRE" ; N1
 40 POKE 32750, INT(N1/256)
 50 POKE 32751, N1 - INT(N1/256)*256
 60 INPUT "DEUXIEME NOMBRE" ; N2
 70 POKE 32752, INT(N2/256)
 80 POKE 32753, N2 - INT(N2/256)*256
 90 EXEC 32701 : PRINT "REPONSE" ;
100 PRINT 65536*PEEK(32762)+256*PEEK(32760)+PEEK(32761)
```

2. Programme assembleur

Lignes	Codes machine	Assembleur
1	FC 7F EE	LDD >32750
2	F3 7F F0	ADDD >32752
3	FD 7F F8	STD >32760
4	86 00	LDA # 0
5	89 00	ADCA # 0
6	B7 7F FA	STA >32762
7	39	RTS

Voici une nouvelle utilisation du bit de retenue qui va nous permettre d'ajouter deux valeurs dont la somme dépasse 65535 et ceci avec, de la part de l'ordinateur, une réponse valable.

Lignes 1, 2 et 3 : l'accumulateur 16 bits D est chargé avec le nombre N1, il lui est ajouté le nombre N2, et le résultat est rangé, sous la forme poids fort/poids faible, dans les octets 32760 et 32761. Le programme pourrait s'arrêter là si nous nous contentions d'ajouter deux nombres ayant une somme plus petite que 65536. Supposons qu'il n'en soit rien et proposons à l'ordinateur le calcul 50000 + 20000 : il va considérer que 70000 se décompose en 65536 d'une part et en 4464 d'autre part. Cette dernière valeur sera écrite dans les octets 32760 et 32761 mais il va garder la trace du débordement de la capacité 16 bits en forçant à 1 le bit de retenue. Il nous faut voir comment nous allons pouvoir nous servir de cette indication.

Lignes 4 et 5 : ces deux lignes ont pour but d'écrire dans le registre A le chiffre du bit de retenue. On met l'accumulateur à 0 et on lui ajoute alors la retenue et la valeur 0. Au total A contiendra bien la valeur d'origine de l'indicateur.

Ligne 6 : il ne reste qu'à ranger ce résultat dans l'octet 32762, là où le programme appelant pourra le retrouver.

En définitive, si le calcul de la somme dépasse 16 bits, le nombre 65536 est ajouté au résultat final par la ligne BASIC 100.

Un petit détail vous aura peut-être échappé : l'instruction LDA #0 de la ligne 4 n'a volontairement pas été remplacée par CLRA. Cette commande, nous l'avons vu, a une action sur le bit de retenue : elle le met toujours à 0. Et, si nous l'avions utilisée, le programme n'aurait pas donné le résultat escompté. Assurez-vous-en en remplaçant les codes machine 86 et 00 par 4F et 12, 12 étant le code de l'instruction NOP (*No OPeration*) qui n'a aucune action sur le déroulement d'un programme. On s'en sert dans les mises au point, pour remplacer des codes machine lorsqu'on s'aperçoit qu'ils sont en trop, par exemple.

ADCB

Le résultat d'une somme entre une valeur 8 bits, le contenu de l'indicateur de retenue et celui de registre B est placé dans ce registre. On peut employer les modes immédiat, indexé et étendu.

ASRA

Tous les bits de l'accumulateur sont décalés sur la droite et le bit 0 va dans l'indicateur de retenue. Mais le bit 7 reste inchangé. Le mode d'adressage inhérent est le seul possible.

Exemple : MODE D'ADRESSAGE INHÉRENT (8 octets)

1. Programme BASIC

```
10
20
30 INPUT "DONNEZ UN NOMBRE NEGATIF" ; N
40 POKE 32750, 256 + N : EXEC 32701
50 PRINT "VOICI SON QUOTIENT PAR DEUX" ;
60 PRINT "–" ; 256–PEEK(32760) : GOTO 30
```

2. Programme assembleur

Lignes	Codes machine	Assembleur	
1	B6 7F EE	LDA	>32750
2	47	ASRA	
3	B7 7F F8	STA	>32760
4	39	RTS	

Il faut rappeler que les nombres 8 bits dont l'écriture binaire commence par le chiffre 1 sont considérés par l'ordinateur comme négatifs. Par exemple −90 s'obtient en calculant le complément à deux de 90, ce qui donne 10100110. Examinons quel sera l'effet de l'instruction ASRA sur ce nombre si l'on suppose qu'il est écrit dans l'accumulateur :

	Accumulateur	C
avant exécution	1 0 1 0 0 1 1 0	?
après exécution	1 1 0 1 0 0 1 1	0

Tous les chiffres ont été décalés sur la droite et le dernier d'entre eux est passé dans l'indicateur. Quant au bit 7, il valait 1 et, dans la place qu'il a laissée libre, le même chiffre 1 a été écrit. En se livrant au jeu des conversions, on obtient pour A la valeur décimale 211. Or, si l'on cherche le complément à deux de 45, on obtient justement 211. Ainsi, à la suite de l'exécution de ASRA, l'accumulateur contient la traduction binaire de la valeur −45. Voici donc compris le rôle de notre nouvelle instruction : elle permet de diviser par 2 un nombre négatif tout en conservant son signe.

Il nous faut voir, au niveau du BASIC, par quelle gymnastique nous pouvons faire parvenir au processeur le nombre à diviser et récupérer ensuite son quotient.

N est un nombre négatif qu'il va falloir transmettre sur le mode complément à deux. Ceci se fait avec le POKE de la ligne 40 : en effet, en retranchant un nombre de 256, on obtient la valeur décimale de son complément à deux. 255 correspond par exemple à −1, 254 à −2, etc.

On reprendra la même méthode pour traduire (ligne 60) le nombre négatif que la machine aura calculé en une forme qui nous est habituelle.

Une dernière chose : ne manquez pas de proposer à l'ordinateur des nombres impairs ou des nombres dont la valeur absolue est supérieure à 127. Et essayez de retrouver à chaque fois où se trouve la logique d'une réponse apparemment incorrecte.

ASRB

C'est cette fois les bits de l'accumulateur B qui sont décalés sur la droite. Le bit 7 reste inchangé et le bit de droite tombe dans l'indicateur de retenue. On utilise cette instruction avec le mode inhérent.

ASR

Le contenu d'un octet mémoire est soumis à une rotation sur sa droite. Le bit 7 garde sa valeur d'origine et le bit 0 passe dans l'indicateur de retenue. On utilise les modes indexé et étendu.

Exemple : MODE D'ADRESSAGE INDEXÉ (28 octets)

1. Programme BASIC

```
10
20
30 EXEC 32701
```

2. Programme assembleur

Lignes	Codes machine	Assembleur		
1	8E 40 00		LDX	#16384
2	86 80		LDA	#128
3	A7 80	SUITE 1	STA	,X+
4	8C 5F 40		CMPX	#24384
5	25 F9		BLO	SUITE1(−7)
6	C6 07		LDB	#7
7	8E 40 00	SUITE 3	LDX	#16384
8	67 80	SUITE 2	ASR	,X+
9	8C 5F 40		CMPX	#24384
10	25 F9		BLO	SUITE2(−7)
11	5A		DECB	
12	26 F3		BNE	SUITE3(−13)
13	39		RTS	

Lignes 1, 2, 3, 4 et 5 : on écrit 128 dans le premier octet écran, ce qui a pour effet d'allumer le point le plus à gauche. On recommence ensuite la même opération pour les 8000 segments de l'écran. A ce moment-là, 40 lignes verticales apparaissent sur notre téléviseur.

Lignes 6 et 7 : on entre dans une boucle qui va à nouveau concerner les 8000 segments écran et ceci, sept fois de suite.

Ligne 8 : voici ce qui se passe dans le segment 16384 :

avant ASR	1	0	0	0	0	0	0	0
après ASR	1	1	0	0	0	0	0	0

Ce sont donc les deux points de gauche qui vont s'allumer. On n'a pas à s'occuper ici du bit de retenue car, avec cette instruction, il n'est pas remis en circulation.

Lignes 9 et 10 : tant que X contiendra une valeur inférieure à 24384, l'ordinateur retournera à la ligne 8, allumant à chaque fois les deux premiers points du segment correspondant.

Ligne 11 : en arrivant à cette instruction, on aura à nouveau quarante lignes sur l'écran, mais chacune d'elles sera large de deux points.

La suite est maintenant facile à deviner. On remonte à SUITE3 et, cette fois, ce seront les trois premiers points de chaque segment qui se verront allumés.

Quand le programme assembleur sera terminé, les huit points de tous les segments vidéo seront visibles et l'écran sera alors entièrement coloré.

Il est possible, cette fois encore, de juger de l'extraordinaire vitesse d'exécution du langage machine : en moins d'une seconde, 8000 octets auront été modifiés huit fois chacun.

COMA

Le contenu de l'accumulateur est remplacé par son complément logique. Chaque chiffre 1 est transformé en un chiffre 0 et réciproquement. Le mode d'adressage inhérent est le seul utilisable.

Exemple : MODE D'ADRESSAGE INHÉRENT (23 octets)

1. Programme BASIC

```
10
20
30 EXEC 32701
```

2. Programme assembleur

Lignes	Codes machine	Assembleur		
1	B6 E7 C3		LDA	> 59331
2	8A 01		ORA	#1
3	B7 E7 C3		STA	> 59331
4	8E 40 00		LDX	#16384
5	A6 84	SUITE	LDA	,X
6	43		COMA	
7	A7 80		STA	,X+
8	8C 5F 40		CMPX	#24384
9	26 F6		BNE	SUITE(−10)
10	35 80		PULS	PC

Voici le programme qui va réaliser l'inversion vidéo de tous les points de l'écran. Si les caractères étaient, par exemple, écrits en bleu sur fond blanc, on les retrouvera écrits en blanc sur fond bleu.

Lignes 1, 2 et 3 : contrairement à ce que l'on pourrait croire au premier abord, nous n'allons à aucun moment agir directement sur la couleur des segments. C'est pour cela que nous nous assurons que l'octet 59331 a son dernier bit à 1.

Lignes 4 et 5 : on charge dans l'accumulateur le contenu de l'octet pointé par X. Dans A est donc inscrite la configuration du premier segment de l'écran.

Lignes 5 et 6 : supposons que A contienne le nombre 00011000 en binaire et que les couleurs en service dans notre programme soient les couleurs standard. Alors les deux points du milieu du segment 16384 s'allumeront en bleu foncé sur bleu clair.

Faisons fonctionner l'instruction COMA :

avant exécution	0	0	0	1	1	0	0	0
après exécution	1	1	1	0	0	1	1	1

Chaque chiffre 1 est devenu un 0 et, inversement, chaque chiffre 0 est devenu un 1. Pour ce qui concerne notre segment, on voit que maintenant ce sont les trois premiers points et les trois derniers qui s'allument en bleu foncé, les deux autres prenant la couleur bleu clair.

Lignes 8 et 9 : l'inversion des couleurs qui a été réalisée pour un segment doit l'être pour tous les autres ; la boucle sera donc exécutée 8000 fois.

Ligne 10 : ne cherchez pas RTS, elle a été remplacée par une nouvelle forme, PULS PC, qui lui est équivalente quant à ses effets. Lorsque l'ordinateur rencontre la commande EXEC, il déconnecte le BASIC et lance l'exécution du programme machine ; mais il prend auparavant le soin de stocker dans la pile système l'adresse à laquelle il devra revenir, une fois exécutés les codes machine. Or, que faisons-nous à la ligne 10 ? Nous ressortons de la pile l'adresse en question et nous la plaçons dans le compteur ordinal PC. Le programme se poursuivra donc à cette adresse. Vous trouvez que l'on s'est compliqué la vie pour rien ? Vous avez raison, la prochaine fois on reprendra RTS.

COMB

Chaque chiffre du registre B est remplacé par son opposé binaire. COMB est une instruction qui ne s'utilise que sur le mode inhérent.

COM

Cette instruction remplace le contenu d'un octet mémoire par son complément logique. On peut employer les adressages indexé et étendu.

Exemple : MODE D'ADRESSAGE INDEXÉ (36 octets)

1. Programme BASIC

```
10
20
30 CLS : PRINT "INVERSION VIDEO DE LA PREMIERE LIGNE"
40 EXEC 32701
```

2. Programme assembleur

Lignes	Codes machine	Assembleur		
1	B6 E7 C3		LDA	> 59331
2	8A 01		ORA	#1
3	B7 E7 C3		STA	> 59331
4	C6 32		LDB	#50
5	34 04	DÉBUT	PSHS	B
6	8E 40 00		LDX	#16384
7	63 80	SUITE	COM	,X+
8	8C 41 40		CMPX	#16704
9	25 F9		BLO	SUITE(-7)
10	CC 27 10		LDD	#10000
11	83 00 01	TEMPO	SUBD	#1
12	26 FB		BNE	TEMPO(-5)
13	35 04		PULS	B
14	5A		DECB	
15	26 E7		BNE	DÉBUT(-25)
16	39		RTS	

Lignes 1, 2 et 3 : le bit 0 de l'octet 59331 est forcé à 1, toujours pour la même raison : c'est la forme du segment qui va entrer en jeu dans notre programme.

Lignes 4 et 5 : B servira de compteur pour la boucle qui sera exécutée 50 fois. On sauvegarde ce registre car sa valeur sera perdue à la ligne 10.

Lignes 6, 7, 8 et 9 : X pointe sur l'adresse du premier segment de l'écran, segment auquel on fait subir une complémentation logique. Il va donc se dessiner sous la forme vidéo inverse. Ce même traitement est effectué sur les 320 premiers segments de l'image, ce qui correspond à la ligne de caractères que nous avons fait écrire par la ligne BASIC 30. A cet instant, la phrase apparaît avec des couleurs inversées par rapport aux précédentes.

Lignes 10, 11 et 12 : un temps mort est créé, la boucle de temporisation tourne à vide 10000 fois.

Lignes 13, 14 et 15 : on retrouve la valeur initiale de B, c'est-à-dire 50 ; on la décrémente et on retourne à la ligne DEBUT où B sera à nouveau mis dans la pile.

Un deuxième passage dans la boucle principale va, cette fois encore, inverser les octets correspondant aux 320 premiers segments de l'écran. Notre phrase retrouvera alors ses couleurs de départ.

Quand le programme arrivera à son terme, les caractères de la première ligne auront clignoté 50 fois.

Naturellement, il est tout à fait envisageable de faire clignoter n'importe quelle ligne de l'écran. Nous engageons le lecteur à rechercher le programme qui serait capable de le faire de lui-même. La ligne BASIC 30 demanderait par exemple :

```
INPUT "QUELLE LIGNE VOULEZ-VOUS VOIR CLIGNOTER" ; L
```

et quatre POKE écriraient dans les octets du programme assembleur les valeurs convenables.

NEGA

La valeur du registre A est remplacée par son complément à deux. NEGA s'utilise en mode d'adressage inhérent.

Exemple : MODE D'ADRESSAGE INHÉRENT (8 octets)

1. Programme BASIC

```
10 CLEAR, 32700 : A$ = "B67FEE40B77FEF39"
20 AD = 32701 : FOR I = 0 TO 7
30 POKE AD + I, VAL("&H"+MID$(A$,2*I+1,2)) : NEXT
40 INPUT "DONNEZ UN NOMBRE" ; N
50 POKE 32750, N : EXEC 32701
60 PRINT "EN COMPLEMENT A DEUX ?" ; -N ;
70 PRINT "S'ECRIT" ; PEEK (32751) : GOTO 40
```

2. Programme assembleur

Lignes	Codes machine	Assembleur	
1	B6 7F EE	LDA	>32750
2	40	NEGA	
3	B7 7F EF	STA	>32751
4	39	RTS	

Nous voici, avec ce programme, débarrassés de tous les problèmes d'écriture des nombres négatifs sur le mode complément à deux. L'instruction NEGA effectue pour nous les deux opérations nécessaires :

— complémentation logique.

— addition de 1 au résultat obtenu.

Ligne 1 : l'accumulateur est chargé avec le nombre N que le BASIC avait écrit dans l'octet 32750.

Ligne 2 : on recherche le complément à deux de N. Cette ligne aurait pu être remplacée par les deux instructions assembleurs suivantes :

```
COMA
ADDA #1
```

Il ne reste ensuite plus qu'à écrire la réponse dans l'octet voulu. On a rencontré peu de programmes machine aussi faciles à comprendre, aussi perdons un peu de temps à analyser la façon dont les codes ont été chargés par le BASIC.

Ligne BASIC 10 : la variable chaîne A$ est formée par la série des codes machine B6, 7F, EE ..., concaténés les uns aux autres.

Ligne BASIC 20 : nous retrouvons notre valeur habituelle 32701, c'est l'adresse à laquelle sera placé le premier code B6. Une boucle FOR NEXT portant sur la variable I est alors exécutée : lorsque I vaut 0, MID$(A$,2*I+1,2) devient MID$(A$,1,2), c'est-à-dire le nombre hexadécimal B6 que POKE placera dans l'octet 32701. Puis I vaudra 1 et, cette fois, POKE inscrira le code 7F dans l'octet 32702. Ceci se poursuivra jusqu'à ce que 39 soit écrit dans l'octet 32708.

Cette méthode est un peu moins lisible que celle que nous avons utilisée tout au long de ce livre, mais, puisque beaucoup de programmeurs la préfèrent, autant l'avoir vue au moins une fois.

NEGB

Le complément à deux du contenu de B est calculé, puis remis dans ce registre. Le mode d'adressage utilisable est l'inhérent.

NEG

Le complément à deux du contenu d'un octet mémoire est effectué. Les modes d'adressage indexé et étendu sont permis.

Exemple : MODE D'ADRESSAGE INDEXÉ (5 octets)

1. Programme BASIC

```
10
20
30 DEFUSR0 = 32701
40 INPUT "DONNEZ UN NOMBRE" ; N%
50 R% = USR0(N%)
60 PRINT "EN COMPLEMENT A DEUX ," ; - N% ;
70 PRINT "S'ECRIT" ; R% : GOTO 40
```

2. Programme assembleur

Lignes	Codes machine	Assembleur	
1	60 03	NEG	3,X
2	86 02	LDA	#2
3	39	RTS	

Le but de ce programme est rigoureusement le même que celui que nous avons vu à la page précédente : on envoie une valeur au microprocesseur, il la complémente à deux et il nous renvoie la réponse. Mais la présentation en est totalement différente : on utilise cette fois la fonction BASIC USR. Un peu plus délicate à mettre en œuvre qu'EXEC, elle a sur celle-ci un avantage très réel : elle permet de faire ce que les informaticiens appelle un "passage de paramètres".

Ligne BASIC 30 : DEFUSR0 sert à indiquer à l'ordinateur à quelle adresse le programme machine numéro 0 doit être lancé. On peut défi-

nir jusqu'à dix sous-programmes avec cette instruction, le dernier étant déterminé par DEFUSR9. Pour notre part, nous n'aurons besoin que d'un seul.

Ligne BASIC 50 : le BASIC appelle au moyen de la fonction USR0 le sous-programme écrit en assembleur. Un peu comme si nous avions donné l'ordre EXEC 32701 mais avec la différence suivante : la valeur N% est transmise au microprocesseur qui la place sur deux octets mémoire, le poids fort dans le premier et le poids faible dans l'autre. Nous ne savons pas précisément où se trouvent ces deux octets, mais l'ordinateur, lui, le sait. Et il le sait grâce au registre X qui pointe sur l'octet situé deux positions avant le premier octet qui nous intéresse.

Hum, pas très clair... Prenons un exemple, on devrait arriver à se comprendre :

Supposons que dans le registre X il y ait le nombre 24917. Cela voudra dire que le poids fort de N% sera situé dans l'octet 24919 et son poids faible dans l'octet 24920. Si, pour N%, nous avons choisi la valeur décimale 10, on trouvera dans le premier de ces octets la valeur 0 (poids fort) et dans l'autre la valeur 10 (poids faible). L'octet 24919 contiendra toujours dans notre programme le chiffre 0 puisque, du fait que l'instruction NEG n'agit que sur un octet, nous ne pouvons proposer à l'ordinateur que des nombres inférieurs à 256, c'est-à-dire des nombres dont le poids fort vaut 0.

La conclusion de tout ceci est que le nombre que nous proposons à la machine est situé dans l'octet 24920. On comprend alors la raison de l'instruction NEG   3,X.

La suite se fait sans notre intervention. Quand le programme s'achève, le microprocesseur retransmet au BASIC les contenus des octets pointés par X + 2 et X + 3. La règle poids fort/poids faible s'applique à nouveau et nous donne la valeur de R%. Notons tout de même que le registre A doit indiquer le type des paramètres retransmis (entier, réel, double précision ou chaîne) et que la valeur 2 correspond au type entier (*Cf.* manuel de référence du BASIC TO 7).

JSR

Cette instruction est l'abréviation de Jump to SubRoutine. *Elle indique au microprocesseur à quelle adresse le sous-programme doit démarrer. Le retour du sous-programme s'effectue à l'instruction qui suit JSR. On peut utiliser les modes d'adressage indexé et étendu.*

Exemple : MODE D'ADRESSAGE ÉTENDU (15 octets)

1. Programme BASIC

```
10
20
30 CLS : EXEC 32701
```

2. Programme assembleur

Lignes	Codes machine	Assembleur		
1	C6 41		LDB	#65
2	34 04	SUITE	PSHS	B
3	BD E8 03		JSR	>59395
4	35 04		PULS	B
5	5C		INCB	
6	C1 5B		CMPB	#91
7	25 F4		BLO	SUITE(−12)
8	39		RTS	

Lignes 1 et 2 : le registre B est chargé avec un nombre qui est le code ASCII de la lettre majuscule A. Ce code est ensuite sauvegardé dans la pile système.

Ligne 3 : on ordonne au microprocesseur de partir exécuter un sous-programme dont l'adresse est écrite dans les octets 59395 et 59396. Ce sous-programme fait partie de la zone mémoire qu'on appelle le moniteur et réalise les fonctions d'affichage d'un caractère sur l'écran.

Quand on effectue un PRINT, par exemple, il est mis en œuvre par le basic sans que, naturellement, nous n'en ayons conscience.

Nous allons nous-mêmes l'utiliser pour faire écrire les 26 lettres de l'alphabet. Ce sous-programme fait toujours apparaître sur l'écran le caractère dont le code ASCII se trouve dans l'accumulateur B. Puisque ce registre contient le nombre 65, c'est donc la lettre A qui va s'écrire en haut et à gauche du téléviseur. Le programme reprendra alors son déroulement normal avec l'instruction PULS B.

Lignes 4, 5 et 6 : on retrouve la valeur initiale de B, on l'incrémente et, puisqu'elle est inférieure à 91, un branchement est effectué à la ligne SUITE.

Après avoir sauvegardé la nouvelle valeur de B, 66, on repart dans le sous-programme d'affichage. Cette fois, c'est la lettre B qui apparaîtra sur l'écran à côté de la précédente.

Et ceci se poursuivra jusqu'à l'affichage de la dernière lettre de l'alphabet Z, lettre dont le code ASCII est 90.

L'étude de ce sous-programme est intéressante car elle permet de programmer, en assembleur, l'apparition de messages sur l'écran. Le lecteur est invité à se familiariser avec ce sous-programme en faisant imprimer les caractères de son choix ; les codes ASCII de ces caractères pourront être par exemple écrits dans des octets consécutifs que le langage machine pourra retrouver.

Et puis, pourquoi ne pas essayer de reconstituer la fonction CLS ? Il suffit au fond de faire imprimer sur toute l'image le code 32, non ?

LBRA

Abréviation de Long BRanch Always, *cette instruction permet de réaliser un branchement long à n'importe quel octet de la mémoire.*

Exemple : MODE D'ADRESSAGE RELATIF (22 octets)

1. Programme BASIC

```
 5 REM RENUM
 8 CLEAR, 31999 : FOR I = 32000 TO 32002 : READI$
 9 POKE I, VAL("&H"+I$) : NEXT
12 FOR I = 32500 TO 32518 : READI$
17 POKE I, VAL("&H"+I$) : NEXT
21 DATA 16,01,F1,CC,00,0A,8E,65,F5,ED,02,C3,00,0A
22 DATA AE,84,8C,00,00,26,F4,39
23 EXEC 32000 : LIST
```

2. Programme assembleur

Lignes	Codes machine	Assembleur		
1	16 01 F1		LBRA	RESTE(+497)
2	CC 00 OA	RESTE	LDD	#10
3	8E 65 F5		LDX	#26101
4	ED 02	SUITE	STD	2,X
5	C3 00 0A		ADDD	#10
6	AE 84		LDX	,X
7	8C 00 00		CMPX	#0
8	26 F4		BNE	SUITE(−12)
9	39		RTS	

Ceci est le programme de renumérotation automatique des lignes : il permet d'obtenir des lignes BASIC écrites de 10 en 10. Démontons-en le mécanisme :

Ligne 1 : les premiers codes machine ont été logés dans les octets 32000, 32001 et 32002 et les autres dans les octets 32500 à 32518. Cette

présentation est tout à fait artificielle : elle ne sert qu'à pouvoir mettre en avant la nouvelle instruction LBRA. Celle-ci procède de la même façon que BRA, elle réalise un branchement inconditionnel, mais avec un déplacement qui s'effectue sur deux octets : aucune difficulté donc pour sauter 497 octets et arriver au cœur du programme.

Lignes 2 et 3 : on charge D avec la valeur 10 ; c'est cet accumulateur qui nous servira à numéroter les lignes de 10 en 10. Puis on place dans X l'adresse du premier octet du programme BASIC, l'octet 26101.

```
FOR I = 26101 TO 26116 : PRINT PEEK(I) ; : NEXT
```

Et voici la réponse :

102 1 0 5 140 32 82 69 78 85 77 0
(première ligne BASIC)
102 32 0 8 .... (début deuxième ligne BASIC)

102 et **1** donnent avec la règle poids fort/poids faible le nombre 26113, c'est-à-dire l'adresse du premier octet de la deuxième ligne.
0 et **5** forment le numéro de la première ligne BASIC.
140 est le code du mot réservé REM.
32, 82, 69, 78, 85 et **77** sont les codes ASCII des caractères qui suivent REM.
0 est le séparateur des deux lignes BASIC.
102 et **32** donnent l'adresse du premier octet de la troisième ligne.
0 et **8** constituent le numéro de la deuxième ligne, etc.

Revenons à l'assembleur. X pointe au départ sur le nombre 102 et ce que nous voulons, c'est remplacer les chiffres 0 et 5 par 0 et 10.

Lignes 4 et 5 : 0 est écrit dans l'octet 26103 et 10 dans le suivant. On ajoute ensuite 10 à D pour avoir le nouveau numéro de la ligne 8.

Ligne 6 : on passe à la deuxième ligne du programme BASIC et X, cette fois, va pointer sur l'octet 26113. C'est le numéro 20 que se verra attribuer cette deuxième ligne.

Le programme bouclera tant que les deux zéros consécutifs qui indiquent la fin du BASIC n'auront pas été rencontrés.

TSTA

Cette instruction teste le contenu de l'accumulateur. Une instruction de branchement doit suivre normalement TSTA. Le seul mode d'adressage que l'on peut utiliser est l'inhérent.

Exemple : MODE D'ADRESSAGE INHÉRENT (35 octets)

1. Programme BASIC

```
 5 CLS : LOCATE 0, 0, 0
10
20
30 LINE (0,199)-(319,100) : EXEC 32701
```

2. Programme assembleur

Lignes	Codes machine	Assembleur		
1	B6 E7 C3		LDA	> 59331
2	8A 01		ORA	#1
3	B7 E7 C3		STA	> 59331
4	8E 40 00		LDX	#16384
5	C6 FF		LDB	#255
6	34 10	DEBUT	PSHS	X
7	E7 84	SUITE	STB	,X
8	30 88 28		LEAX 40	,X
9	A6 84		LDA	,X
10	4D		TSTA	
11	27 F6		BEQ	SUITE(-10)
12	35 10		PULS	X
13	30 01		LEAX 1	,X
14	8C 40 28		CMPX	#16424
15	25 EB		BLO	DEBUT(-21)
16	39		RTS	

Lignes 1, 2 et 3 : puisque la commande CLS a été exécutée, ces trois premières lignes sont indispensables.

Lignes 4, 5 et 6 : B va servir à déterminer la configuration des segments : ils auront tous leurs huit points allumés. X, pour sa part, est chargé avec l'adresse du premier octet écran, et sa valeur est empilée aussitôt.

Ligne 7 : on allume la totalité du premier segment de l'écran.

Ligne 8 : comme 40 est ajouté au registre X, celui-ci va pointer sur le segment 16424 situé exactement au-dessous du précédent.

Lignes 9, 10 et 11 : ce segment, pour l'instant, a une valeur nulle puisqu'aucun de ses points n'est allumé. L'instruction TSTA va donc trouver dans le registre A le chiffre 0 et l'ordinateur sera renvoyé à la ligne 7. Cette fois, le segment se verra allumé et X pointera alors sur l'octet 16464. Quand le programme sortira de la boucle SUITE, une colonne apparaîtra sur la gauche de l'écran. Sa borne inférieure aura été atteinte quand TSTA aura trouvé dans un octet une valeur différente de 0, c'est-à-dire quand la ligne tracée par l'instruction LINE aura été rencontrée.

Lignes 12, 13, 14 et 15 : on ressort de la pile la valeur initiale de X, 16384, on y ajoute 1, on vérifie que l'on n'atteint pas la limite droite de l'image et on rebranche le programme à la ligne 6. Après avoir sauvegardé l'adresse du deuxième segment de l'écran, on retrouve la boucle SUITE : une nouvelle colonne à droite de la précédente va alors se dessiner sur le téléviseur. Sa limite inférieure sera, bien entendu, le segment tracé par la ligne BASIC 30.

En définitive, quand le programme exécutera l'instruction RTS, 40 colonnes seront visibles ; et elles auront toutes pour limite inférieure la même droite.

Une remarque avant de terminer : il est nécessaire de faire disparaître le curseur dans cet exemple car le programme risquerait d'être trompé en testant l'octet qui lui correspond. D'où l'instruction LOCATE de la ligne BASIC 5.

TSTB

C'est dans ce cas le contenu du registre B qui est testé. Cette instruction ne s'emploie qu'avec le mode inhérent.

TST

Cette instruction teste le contenu d'un octet mémoire. Un branchement doit normalement être effectué après ce test. On peut employer les modes d'adressage indexé et étendu.

Exemple : MODE D'ADRESSAGE INDEXÉ (25 octets)

1. Programme BASIC

```
10
20
30 CLS : FOR Y = 24 TO 0 STEP -1 : X = INT(RND*30) + 10
40 LOCATE X, Y, 0 : PRINT "*" ; : NEXT
50 POKE 59331, PEEK(59331) OR 1 : EXEC 32701
```

2. Programme assembleur

Lignes	Codes machine	Assembleur		
1	86 FF		LDA	#255
2	8E 40 A0		LDX	#16544
3	34 10	LIGNE	PSHS	X
4	A7 80	SUITE	STA	,X+
5	6D 84		TST	,X
6	27 FA		BEQ	SUITE(-6)
7	35 10		PULS	X
8	30 89 01 40		LEAX 320	,X
9	8C 5F 3C		CMPX	#24384
10	25 ED		BLO	LIGNE(-19)
11	39		RTS	

Le programme BASIC fait apparaître à chaque ligne de l'écran une étoile dont le numéro de colonne est tiré au sort. Vous avez certainement remarqué que, pour éviter d'avoir à écrire les trois lignes assem-

bleur qui forcent le bit de droite de l'octet 59331 à 1, nous avons commandé au BASIC de le faire. C'est la raison du POKE de la ligne 50.

Lignes 1, 2 et 3 : les deux registres A et X sont initialisés, le premier avec le nombre correspondant à un segment entièrement allumé, et le second avec l'adresse d'un segment situé à gauche de l'écran et quatre positions en dessous de l'origine (16544 = 16384 + 160). Le contenu de X est mis ensuite dans la pile système.

Lignes 4, 5 et 6 : le segment 16544 s'allume et X, incrémenté, passe à 16545. On teste alors le contenu de cet octet : comme il est nul, on allume le segment correspondant et on s'intéresse à celui placé sur sa droite. En fin de compte, on ne sortira de la boucle SUITE, que lorsque l'on aura rencontré un segment déjà allumé, c'est-à-dire lorsque le programme aura trouvé une étoile. Voici expliqué pourquoi une ligne est tracée entre la gauche de l'écran et la première étoile.

Lignes 7, 8, 9 et 10 : on recherche la valeur d'origine de X pour lui ajouter le nombre 320. Du fait que 320 est égal à 8 fois 40, le programme va cette fois concerner une zone écran située 8 points en dessous de la précédente. Prenez deux minutes pour comprendre comment l'instruction LEAX 320,X a été codée sur 4 octets et puis nous aborderons la suite.

Le programme donc retourne à la troisième instruction et sauvegarde la nouvelle valeur du registre X. Tous les segments d'une même ligne horizontale vont alors, en partant de la gauche, s'allumer et ceci jusqu'à qu'une étoile soit rencontrée. C'est ce qui correspond à la deuxième droite qui apparaît sur l'écran.

Quand le programme assembleur est achevé, nous pouvons voir sur notre téléviseur 25 lignes qui, partant toutes de la gauche, sont tracées jusqu'à ce qu'elles touchent une étoile.

A titre d'exercice, le lecteur pourra essayer de concevoir un programme équivalent à celui-ci, mais il s'agira cette fois de droites verticales qui devront partir du haut de l'écran.

EXG

Cette instruction permet d'échanger les contenus de deux registres de même dimension. On l'utilise avec le mode d'adressage inhérent.

Exemple : MODE D'ADRESSAGE INHÉRENT (17 octets)

1. Programme BASIC

```
10
20
30 CLS : POKE 59331, PEEK(59331) OR 1 : EXEC 32701
```

2. Programme assembleur

Lignes	Codes machine	Assembleur		
1	86 FF		LDA	#255
2	C6 10		LDB	#16
3	8E 40 00		LDX	#16384
4	ED 81	SUITE	STD	,X++
5	1E 89		EXG	A,B
6	8C 5F 40		CMPX	#24384
7	26 F7		BNE	SUITE(−9)
8	39		RTS	

Lignes 1, 2 et 3 : A et B vont servir à imprimer des segments sur l'écran : A correspond à un segment entièrement éclairé et B à un segment dont seul un point sera visible. X contient, cette fois encore, l'adresse du premier octet écran.

Ligne 4 : on range l'accumulateur D dans les octets 16384 et 16385. D étant constitué des registres A et B, cette instruction est rigoureusement équivalente aux deux suivantes :

```
– STA ,X+
– STB ,X+
```

Nous voyons donc s'allumer les huit points du premier segment et le point central du deuxième.

Ligne 5 : voici l'instruction qui échange les accumulateurs 8 bits ; le premier, A, va contenir la valeur 16 et le second, B, la valeur 255.

Lignes 6 et 7 : X a été incrémenté deux fois à la ligne 4 et il pointe donc sur l'octet 16386. On vérifie que l'on ne sort pas de l'image, et on rebranche le programme à la ligne SUITE.

C'est dans les octets 16386 et 16387 que D est cette fois placé ; son poids fort dans le premier et son poids faible dans l'autre. Un point suivi d'un segment vont donc s'éclairer. Les accumulateurs A et B seront à ce moment-là échangés et ceci se poursuivra jusqu'à la fin du programme. Du fait que deux points succèdent toujours à deux traits, nous verrons apparaître sur notre écran une succession de colonnes et de lignes verticales. Les colonnes auront une largeur correspondant à deux segments et les lignes iront par paires.

Intéressons-nous maintenant à la façon dont les codes machine de l'instruction EXG s'obtiennent :

D	X	Y	U	S	PC	A	B
0	1	2	3	4	5	8	9

1E est le code réservé à la commande EXG et les deux chiffres 8 et 9 correspondent aux registres que nous voulons échanger. On aurait, cela va de soit, pu coder la ligne 5 par 1E 98.

Autre exemple : si nous avions souhaité échanger les contenus des registres X et Y, nous aurions employé le codage 1E 12. On a vu, dans ce livre, des notions bien plus difficiles à comprendre !

TFR

TFR est une instruction qui permet de transférer le contenu d'un premier registre dans un deuxième. Le transfert ne peut se réaliser que sur des registres de même dimension. On emploie le mode d'adressage inhérent. On code les registres de la même façon qu'avec EXG.

Exemple : MODE D'ADRESSAGE INHÉRENT (41 octets)

1. Programme BASIC

```
10
20
30 CLS : POKE 59331, PEEK(59331) OR 1
40 POKE 32750, 40 : EXEC 32701
```

2. Programme assembleur

Lignes	Codes machine	Assembleur		
1	8E 00 00		LDX	#0
2	86 20		LDA	#32
3	C6 7F		LDB	#127
4	BD E8 03	AFFICH	JSR	> 59395
5	34 04		PSHS	B
6	1F 89		TFR	A,B
7	35 02		PULS	A
8	7A 7F EE		DEC	> 32750
9	26 0A		BNE	SUITE(+10)
10	1E 89		EXG	A,B
11	10 8E 00 28		LDY	#40
12	10 BF 7F ED		STY	> 32749
13	30 01	SUITE	LEAX	1,X
14	8C 03 E7		CMPX	#999
15	26 E1		BNE	AFFICH(−31)
16	7E FF F8		JMP	> 65528

Lignes 1, 2 et 3 : le registre X est mis à zéro ; c'est lui qui comptera les 999 cases de l'écran. Les accumulateurs A et B sont chargés respectivement avec les codes ASCII de la case vide — CHR$(32) — et de la case pleine — CHR$(127).

Ligne 4 : branchement vers le sous-programme d'affichage. Puisque B contient la valeur 127, un petit carré va s'imprimer en haut et à gauche du téléviseur.

Lignes 5, 6 et 7 : on sauvegarde la valeur du registre B, soit 127, dans la pile système S. On transfère le contenu de A dans B : à ce moment précis, les deux registres contiennent le même nombre 32. Puis on ressort de la pile le nombre qui s'y trouvait pour l'écrire dans A ; ce qui fait que A contient maintenant la valeur 127. Ces trois lignes sont donc équivalentes à EXG A,B. Leur seul intérêt est de mettre en application l'instruction de transfert.

Lignes 8 et 9 : on retranche 1 au contenu de l'octet 32750. Comme c'est le nombre 40 que nous y avions placé par POKE, l'instruction BNE va exécuter un branchement à la ligne 13 du programme. Là, on rajoute 1 au registre X et on retourne au sous-programme d'affichage. Puisque B vaut 32, c'est le caractère espace qui sera imprimé dans la deuxième case de l'écran. Un nouvel échange des accumulateurs va alors s'effectuer et au troisième passage de la boucle AFFICH, c'est une case pleine qui apparaîtra. L'alternance case foncée/case claire explique donc l'effet de damier obtenu.

Reste à voir les détails :

Lignes 10, 11 et 12 : quand l'octet 32750 arrive à zéro, cela correspond au fait que la droite de l'écran est atteinte. On procède alors à un nouvel échange des registres A et B pour que la deuxième ligne débute par une case claire. Le programme respectera alors la structure d'un damier. Il ne reste plus ensuite qu'à remettre au niveau 40 l'octet 32750 pour que le passage de la deuxième à la troisième ligne se fasse de façon correcte.

Quant à l'instruction écrite à la dernière ligne, elle ne fait pas autre chose que RTS. JMP est en effet une instruction de branchement : elle provoque, dans notre cas, l'envoi de l'ordinateur à l'adresse 65528. Et qu'y a-t-il de programmé dans cet octet ? Le retour au BASIC, tout simplement.

CONCLUSION

Ce livre a constitué une introduction à la programmation en langage machine de l'ordinateur TO 7. Nous en avons étudié les aspects les plus importants et réalisé une série d'exercices qui vous ont montré, c'est notre souhait, que l'assembleur pouvait être assimilé sans difficulté par un lecteur armé de sa seule bonne volonté. Nous sommes persuadés, pour notre part, qu'il est infiniment plus long d'acquérir la logique de la programmation BASIC que celle de l'assembleur.

Vous êtes maintenant en mesure de créer vos propres programmes et d'inclure dans vos lignes BASIC des effets spéciaux que seule l'impressionnante rapidité de l'assembleur autorise. Si l'occasion se présente, vous ne manquerez pas de chercher à quoi correspondent les codes machine que d'autres programmeurs auront obtenus, faisant ainsi le travail inverse de celui qui a été effectué jusqu'à maintenant. Cette opération, qui s'appelle le désassemblage, vous permettra de reconstruire le programme assembleur et éventuellement de le modifier pour qu'il s'adapte très précisément à votre cas.

Naturellement, rien ne vous empêche de franchir une nouvelle étape en vous orientant vers des ouvrages plus techniques que celui-ci[1]. Vous y trouverez des programmes applicables à la gestion des périphériques ainsi que des explications concernant les quelques instructions que nous avons volontairement passées sous silence, estimant que, dans un premier temps en tout cas, leur intérêt était négligeable.

Ce livre s'achève sur trois programmes un peu plus compliqués que les autres. Vous les aborderez sans complexes maintenant que vous est ouvert l'étroit mais ô combien royal chemin de l'assembleur.

1. *Programmation du 6809* par Rodnay Zaks et William Labiak.

ANNEXE 1
GESTION DE L'ERREUR I/0

1. Programme BASIC

```
10 CLEAR, 32699 : FOR I = 32701 TO 32745 : READ I$ :
   POKE I, VAL ("&H" + I$ ) : NEXT
20 DATA .... (45 codes machine)
```

2. Programme assembleur

Lignes	Codes machine	Assembleur		
1	8E 65 FE		LDX	#26110
2	A6 80	SUITE	LDA	,X+
3	81 00		CMPA	#0
4	26 FA		BNE	SUITE(−6)
5	BF 65 F5		STX	>26101
6	7F 65 F7		CLR	>26103
7	7F 65 F8		CLR	>26104
8	8E 65 F5		LDX	#26101
9	AE 84	SUITE2	LDX	,X
10	BF 7F EE		STX	>32750
11	EC 84		LDD	,X
12	B3 7F EE		SUBD	>32750
13	10 83 01 00		CMPD	#256
14	25 F0		BLO	SUITE2(−16)
15	10 8E 00 00		LDY	#0
16	10 AF 84		STY	,X
17	39		RTS	

I/0 Error, le message tant redouté des possesseurs du TO 7, est apparu sur votre écran. Voici le programme qui va permettre, en partie du moins, de remédier à vos ennuis.

Les modalités à accomplir sont les suivantes :

— Tapez les lignes 10 et 20 du programme BASIC
— Exécutez ce programme (RUN)
— Chargez avec LOAD le programme défectueux jusqu'au point d'erreur
— Tapez en mode direct : EXEC 32701

A ce moment-là, la commande LIST va faire apparaître sur l'écran la partie du programme correspondant à ce que l'ordinateur a pu charger avant l'arrêt du magnétophone. Vous n'aurez naturellement jamais le listing complet, mais c'est déjà mieux que rien, n'est-ce pas ?

L'assembleur aura débloqué la commande LIST, mais elle seulement. Inutile donc d'essayer RUN ou SAVE, par exemple ; vous ne feriez que "planter" votre ordinateur. Puisque LIST est utilisable, sauvegardez ce qui est en mémoire avec l'instruction LIST "CASS:XXX", XXX étant le nom supposé de votre programme. Eteignez alors le TO 7 pour qu'il se réinitialise, rallumez-le et replacez en mémoire (avec LOAD) le programme que vous aurez récupéré. Après vérification de la première et de la dernière lignes, vous pourrez retaper les lignes manquantes.

Avant d'étudier la logique de la méthode, revoyons ce que contiennent les premiers octets de la mémoire BASIC.

Supposons que l'on a tapé les lignes suivantes :

```
10 REMABCDEF
20 .....
```

Si le programme est chargé correctement, voici ce que contiennent les octets 26101 à 26116 :

102 1 0 10 140 65 66 67 68 69 70 0 102 43 0 20

102 et 1 correspondent à l'adresse de l'octet qui débute la deuxième ligne soit, pour nous, 26113.

Si une erreur I/0 a été décelée, voici ce que contiennent ces mêmes octets :

0 0 64 73 143 75 246 0 0 69 70 0 102 43 0 20

Seuls les neuf premiers octets ont été modifiés. A notre charge de remettre de l'ordre dans cela avec l'assembleur.

Lignes 1, 2, 3 et 4 : on charge X avec l'adresse du premier octet correct, celui qui donc vaut 69. On en fait passer le contenu dans A et, si ce registre n'est pas nul, on incrémente X puis on s'intéresse à l'octet suivant. La boucle SUITE va ainsi être effectuée jusqu'à ce que le chiffre 0 soit rencontré. La valeur nulle, placée dans l'octet 26112 de notre exemple, est en effet le séparateur de deux lignes BASIC.

Lignes 5, 6 et 7 : X, venant d'être incrémenté, contient le nombre 26113, c'est-à-dire très exactement le nombre que devraient contenir les octets 26101 et 26102. STX est donc là pour écrire dans ces deux octets les valeurs 102 et 1. On fait ensuite en sorte, en rendant nuls les octets 26103 et 26104 que la première ligne BASIC ait pour numéro 0. Peu nous importent les octets 26105 à 26109, puisque, si nécessaire, nous pourrons à la fin de toutes les opérations, réécrire correctement la première ligne de notre programme.

Lignes 8 à 14 : on vient de s'occuper du déblocage de la commande LIST, mais il est obligatoire de trouver maintenant à quel octet de la mémoire s'arrête la partie du programme que nous avons récupérée.

26101 est placé dans X, et immédiatement après, LDX, X charge ce registre avec le nombre 26113. Cette valeur est alors stockée temporairement dans l'octet 32750 et, dans l'accumulateur D, est écrite la valeur pointée par X soit 26155 (obtenue par la règle poids fort/poids faible appliquée aux nombres 102 et 43). On retranche ensuite 26113 de 26155 et on compare cette différence avec 256. Tant que l'on sera dans le programme BASIC, un branchement à SUITE2 sera forcément réalisé puisqu'une ligne BASIC ne peut contenir que 255 caractères au maximum. Ce branchement placera 26155 dans X et l'adresse du premier octet de la quatrième ligne dans D.

Le programme bouclera jusqu'à ce que soit atteinte la zone mémoire défectueuse. A ce moment-là, D contiendra un nombre absolument quelconque et l'assembleur reconnaîtra que cette valeur ne respecte pas la règle du nombre d'octets maximum que peut contenir une ligne BASIC. Il ne restera plus qu'à écrire le chiffre 0 dans les deux octets pointés par X et ceci pour indiquer à la commande LIST que c'est la fin du programme.

ANNEXE 2
TRI EN MÉMOIRE CENTRALE

1. Programme BASIC (48 octets)

```
10
20
30 DEFINT A-Z : DIMA(99)
40 FOR I = 0 TO 99 : A(I) = INT(RND*1000) + 1 : NEXT
50 DEFUSR0 = 32701 : Z = USR0(VARPTR(A(0)))
60 FOR I = 0 TO 99 : PRINT A(I) ; : NEXT
```

2. Programme assembleur

Lignes	Codes machine	Assembleur		
1	86 64		LDA	#100
2	C6 64		LDB	#100
3	10 AE 02		LDY	2,X
4	34 06	SUITE2	PSHS	D
5	30 22		LEAX	2,Y
6	6A E4	SUITE1	DEC	,S
7	27 15		BEQ	PASSE(+21)
8	EC 81		LDD	,X++
9	10 A3 A4		CMPD	,Y
10	24 F5		BHS	SUITE1(−11)
11	34 06		PSHS	D
12	EC A4		LDD	,Y
13	ED 83		STD	,− −X
14	35 06		PULS	D
15	ED A4		STD	,Y
16	30 02		LEAX	2,X
17	20 E7		BRA	SUITE1(−25)
18	31 22	PASSE	LEAY	2,Y
19	35 06		PULS	D
20	5A		DECB	
21	1F 98		TFR	B,A
22	C1 01		CMPB	#1
23	26 D8		BNE	SUITE2(−40)
24	39		RTS	

100 nombres compris entre 1 et 1000 sont choisis aléatoirement et placés dans un tableau A. Le but du programme assembleur est de trier les 100 valeurs et de les recopier, rangées par ordre croissant, dans le tableau. A(0) sera donc l'élément minimum et A(99) l'élément maximum.

Lignes 1 et 2 : les deux accumulateurs contiennent le nombre d'éléments à trier. Ce sont ces deux lignes, et elles seulement, qu'il faut modifier si l'on utilise ce programme pour trier un tableau qui ne contiendrait pas 100 éléments.

Ligne 3 : revoyons les particularités de USR0 : le paramètre que nous faisons passer au langage machine est placé dans deux octets après l'adresse contenue dans X. Ce qui fait que, pour notre exemple, Y va contenir VARPTR(A(0)). Le programme démarre donc avec un registre Y pointant sur le premier élément A(0) du tableau.

Lignes 4 et 5 : l'accumulateur D est sauvegardé et on commande à X de pointer sur le deuxième élément du tableau, donc sur A(1) ; n'oublions pas que les entiers sont rangés en mémoire sur deux octets.

Lignes 6 à 10 : on décrémente l'octet écrit au sommet de la pile. Cet octet correspond au nombre de passages dans la boucle SUITE1. On compare alors le deuxième élément du tableau (pointé par X) et le premier (pointé par Y). Si le deuxième nombre est supérieur ou égal au premier, on ne modifie rien et le programme retourne à SUITE1 pour, cette fois, comparer le troisième et le premier éléments.

Lignes 11 à 17 : si, en revanche, le deuxième nombre est inférieur au premier, on procède à leur échange. Nous sommes donc sûrs, à la fin de la boucle SUITE1, que A(0) est l'élément minimum du tableau.

Lignes 18 et suivantes : Y va contenir l'adresse de A(1) et un retour à SUITE2 est programmé. Pour, cette fois-ci, comparer A(1) avec les éléments suivants ; un échange sera effectué à chaque fois que l'on aura trouvé un élément plus petit que A(1).

Ainsi, après deux passages de la boucle SUITE2, A(0) et A(1) auront les deux plus petites valeurs du tableau, et ceci dans l'ordre croissant. Quand le programme arrivera à son terme, tous les éléments seront rangés correctement et la ligne BASIC 60 nous en donnera la confirmation.

Notons pour finir que la variable Z est une variable fictive dont la valeur nous importe peu puisque l'assembleur n'a aucun paramètre à retourner au BASIC.

ANNEXE 3
DÉPLACEMENT D'UN MISSILE

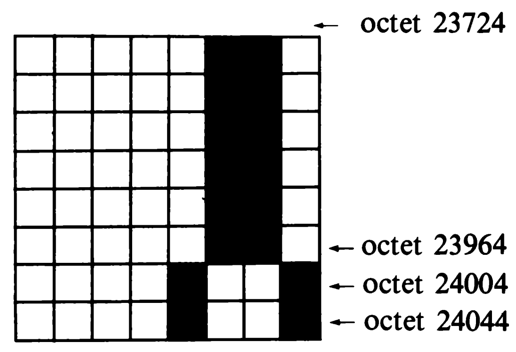
1. Programme BASIC

```
10 CLEAR, 32700, 3 : FOR I = 32701 TO 32761 ....
20 DATA .....
30 DEFGR$(0)=60,60,126,254,127,126,60,60
40 DEFGR$(1)=6,6,6,6,6,6,9,9
50 DEFGR$(2)=64,96,112,126,126,112,96,64
60 SCREEN 1,6,6 : CLS : ATTRB 1,0 : COLOR 7
70 FOR I = 0 TO 19 : PRINT GR$(0);:NEXT
80 COLOR 1 : LOCATE 20,24 : PRINT CHR$(127) ;
90 ATTRB 0,0 : LOCATE 20,23 : PRINT GR$(1)
100 I = INT(RND*3000) : FOR J = 1 TO I : NEXT
110 ATTRB 1,0 : FOR I = 0 TO 19
120 LOCATE 2*I,1 : PRINT GR$(2)
130 IF INKEY$ < > "" THEN 200
140 PLAY "L5P" : LOCATE 2*I,1 : PRINT SPC(1)
150 NEXT : GOTO 90
200 POKE 59331, PEEK(59331) OR 1 : EXEC 32701
210 ATTRB 1,1 : COLOR 0 : LOCATE 0,20
220 IF PEEK(32766)=&H42 THEN PRINT "GAGNE"
    ELSE PRINT "PERDU"
230 PLAY"L96P" : LOCATE 0,20 : PRINT SPC(5)
240 ATTRB 1,0 : LOCATE 2*I,1 : PRINT SPC(1)
250 COLOR 1 : GOTO 90
```

2. Programme assembleur

Lignes	Codes machine	Assembleur		
1	86 09		LDA	#9
2	C6 06		LDB	#6
3	8E 5C AC		LDX	#23724
4	6D 84	TEST	TST	,X
5	26 11		BNE	FIN(+17)
6	E7 84		STB	,X
7	6F 89 01 40		CLR	320,X
8	A7 89 00 F0		STA	240,X
9	30 88 D8		LEAX	−40,X
10	8D 16		BSR	TEMPO(+22)
11	20 EB		BRA	TEST(−21)
12	C6 08	FIN	LDB	#8
13	30 89 01 40		LEAX	320,X
14	6F 84	EFFACE	CLR	,X
15	8D 0A		BSR	TEMPO(+10)
16	30 88 D8		LEAX	−40,X
17	5A		DECB	
18	26 F6		BNE	EFFACE(−10)
19	BF 7F FE		STX	> 32766
20	39		RTS	
21	34 06	TEMPO	PSHS	D
22	CC 01 F4		LDD	#500
23	83 00 01	SUITE	SUBD	#1
24	26 FB		BNE	SUITE(−5)
25	35 06		PULS	D
26	39		RTS	

Voici un exemple de programme assembleur gérant le déplacement vers le haut d'un missile. Sa position de départ et sa forme sont définis par le schéma suivant :



Lignes 1 à 5 : B correspond au corps du missile et A à ses ailerons. Quant à X, il contient l'adresse du segment situé exactement au-dessus du missile. Dès que ce missile aura rencontré un obstacle, X ne contiendra plus 0 et un branchement sera effectué à la fin du programme.

Lignes 6 à 11 : on suppose que la voie est libre devant le missile et on le déplace d'une position vers le haut. Pour cela, on allume deux points de l'octet 23724, on efface le segment 24044 et on dessine les deux ailerons (registre A) dans l'octet 23964. On fait ensuite pointer X sur le segment placé au-dessus du numéro 23724. Une perte de temps est alors programmée et l'ordinateur retourne à la ligne 4. La boucle TEST est donc effectuée tant que le missile ne rencontre ni une fusée ni un nuage.

Lignes 12 à 18 : un obstacle a été percuté ; il nous faut effacer l'une après l'autre les 8 parties qui composent le missile. Puisqu'on veut les faire disparaître en partant du bas, on place dans X (ligne 13) l'adresse de la partie inférieure. Après 8 passages de la boucle EFFACE, le missile aura disparu de l'écran.

Lignes 19 et 20 : on prend soin de noter dans l'octet 32766 l'adresse du segment correspondant à la collision. Cette adresse est égale à 17004 si la fusée a été touchée et à 16684 si c'est un nuage qui a été rencontré. Les poids forts de 17004 et de 16684 valant, en hexadécimal, 42 et 41, cette différence permettra au BASIC de contrôler si la cible a été atteinte ou pas.

APPENDICE A

JEU D'INSTRUCTIONS DU 6809

Instruction	Forms	Addressing Modes															Description	5	3	2	1	0
		Immediate			Direct			Indexed			Extended			Inherent				H	N	Z	V	C
		Op	~	#	Op	~	#	Op	~	#	Op	~	#	Op	~	#						
ABX														3A	3	1	B + X → X (Unsigned)	•	•	•	•	•
ADC	ADCA	89	2	2	99	4	2	A9	4 +	2 +	B9	5	3				A + M + C → A	↕	↕	↕	↕	↕
	ADCB	C9	2	2	D9	4	2	E9	4 +	2 +	F9	5	3				B + M + C → B	↕	↕	↕	↕	↕
ADD	ADDA	8B	2	2	9B	4	2	AB	4 +	2 +	BB	5	3				A + M → A	↕	↕	↕	↕	↕
	ADDB	CB	2	2	DB	4	2	EB	4 +	2 +	FB	5	3				B + M → B	↕	↕	↕	↕	↕
	ADDD	C3	4	3	D3	6	2	E3	6 +	2 +	F3	7	3				D + M M + 1 → D	•	↕	↕	↕	↕
AND	ANDA	84	2	2	94	4	2	A4	4 +	2 +	B4	5	3				A Λ M → A	•	↕	↕	0	•
	ANDB	C4	2	2	D4	4	2	E4	4 +	2 +	F4	5	3				B Λ M → B	•	↕	↕	0	•
	ANDCC	1C	3	2													CC Λ IMM → CC					7
ASL	ASLA													48	2	1	A	8	↕	↕	↕	↕
	ASLB													58	2	1	B ← 0	8	↕	↕	↕	↕
	ASL				08	6	2	68	6 +	2 +	78	7	3				M c b7 b0	8	↕	↕	↕	↕
ASR	ASRA													47	2	1	A	8	↕	↕	•	↕
	ASRB													57	2	1	B	8	↕	↕	•	↕
	ASR				07	6	2	67	6 +	2 +	77	7	3				M b7 b0 c	8	↕	↕	•	↕
BIT	BITA	85	2	2	95	4	2	A5	4 +	2 +	B5	5	3				Bit Test A (M Λ A)	•	↕	↕	0	•
	BITB	C5	2	2	D5	4	2	E5	4 +	2 +	F5	5	3				Bit Test B (M Λ B)	•	↕	↕	0	•
CLR	CLRA													4F	2	1	0 → A	•	0	1	0	0
	CLRB													5F	2	1	0 → B	•	0	1	0	0
	CLR				0F	6	2	6F	6 +	2 +	7F	7	3				0 → M	•	0	1	0	0
CMP	CMPA	81	2	2	91	4	2	A1	4 +	2 +	B1	5	3				Compare M from A	8	↕	↕	↕	↕
	CMPB	C1	2	2	D1	4	2	E1	4 +	2 +	F1	5	3				Compare M from B	8	↕	↕	↕	↕
	CMPD	10	5	4	10	7	3	10	7 +	3 +	10	8	4				Compare M M + 1 from D	•	↕	↕	↕	↕
		83			93			A3			B3											
	CMPS	11	5	4	11	7	3	11	7 +	3 +	11	8	4				Compare M M + 1 from S	•	↕	↕	↕	↕
		8C			9C			AC			BC											
	CMPU	11	5	4	11	7	3	11	7 +	3 +	11	8	4				Compare M M + 1 from U	•	↕	↕	↕	↕
		83			93			A3			83											
	CMPX	8C	4	3	9C	6	2	AC	6 +	2 +	BC	7	3				Compare M M + 1 from X	•	↕	↕	↕	↕
	CMPY	10	5	4	10	7	3	10	7 +	3 +	10	8	4				Compare M M + 1 from Y	•	↕	↕	↕	↕
		8C			9C			AC			BC											
COM	COMA													43	2	1	$\overline{A}$ → A	•	↕	↕	0	1
	COMB													53	2	1	$\overline{B}$ → B	•	↕	↕	0	1
	COM				03	6	2	63	6 +	2 +	73	7	3				$\overline{M}$ → M	•	↕	↕	0	1
CWAI		3C	≥20	2													CC Λ IMM → CC Wait for Interrupt					7
DAA														19	2	1	Decimal Adjust A	•	↕	↕	0	↕
DEC	DECA													4A	2	1	A 1 → A	•	↕	↕	↕	•
	DECB													5A	2	1	B 1 → B	•	↕	↕	↕	•
	DEC				0A	6	2	6A	6 +	2 +	7A	7	3				M 1 → M	•	↕	↕	↕	•
EOR	EORA	88	2	2	98	4	2	A8	4 +	2 +	B8	5	3				A ⊻ M → A	•	↕	↕	0	•
	EORB	C8	2	2	D8	4	2	E8	4 +	2 +	F8	5	3				B ⊻ M → B	•	↕	↕	0	•
EXG	R1 R2													1E	8	2	R1 → R2[2]	•	•	•	•	•
INC	INCA													4C	2	1	A + 1 → A	•	↕	↕	↕	•
	INCB													5C	2	1	B + 1 → B	•	↕	↕	↕	•
	INC				0C	6	2	6C	6 +	2 +	7C	7	3				M + 1 → M	•	↕	↕	↕	•
JMP					0E	3	2	6E	3 +	2 +	7E	4	3				EA[3] → PC	•	•	•	•	•
JSR					9D	7	2	AD	7 +	2 +	BD	8	3				Jump to Subroutine	•	•	•	•	•
LD	LDA	86	2	2	96	4	2	A6	4 +	2 +	B6	5	3				M → A	•	↕	↕	0	•
	LDB	C6	2	2	D6	4	2	E6	4 +	2 +	F6	5	3				M → B	•	↕	↕	0	•
	LDD	CC	3	3	DC	5	2	EC	5 +	2 +	FC	6	3				M M + 1 → D	•	↕	↕	0	•
	LDS	10	4	4	10	6	3	10	6 +	3 +	10	7	4				M M + 1 → S	•	↕	↕	0	•
		CE			DE			EE			FE											
	LDU	CE	3	3	DE	5	2	EE	5 +	2 +	FE	6	3				M M + 1 → U	•	↕	↕	0	•
	LDX	8E	3	3	9E	5	2	AE	5 +	2 +	BE	6	3				M M + 1 → X	•	↕	↕	0	•
	LDY	10	4	4	10	6	3	10	6 +	3 +	10	7	4				M M + 1 → Y	•	↕	↕	0	•
		8E			9E			AE			BE											
LEA	LEAS							32	4 +	2 +							EA[3] → S	•	•	•	•	•
	LEAU							33	4 +	2 +							EA[3] → U	•	•	•	•	•
	LEAX							30	4 +	2 +							EA[3] → X	•	•	↕	•	•
	LEAY							31	4 +	2 +							EA[3] → Y	•	•	↕	•	•

Legend

OP	Operation Code (Hexadecimal)	$\overline{M}$	Complement of M	↕	Test and set if true, cleared otherwise
~	Number of MPU Cycles	→	Transfer Into	•	Not Affected
#	Number of Program Bytes	H	Half-carry (from bit 3)	CC	Condition Code Register
+	Arithmetic Plus	N	Negative (sign bit)		Concatenation
−	Arithmetic Minus	Z	Zero result	V	Logical or
•	Multiply	V	Overflow, 2's complement	Λ	Logical and
		C	Carry from ALU	⊻	Logical Exclusive or

Courtesy of Motorola, Inc.

Instruction	Forms	Immediate Op	Immediate ~	Immediate #	Direct Op	Direct ~	Direct #	Indexed[1] Op	Indexed ~	Indexed #	Extended Op	Extended ~	Extended #	Inherent Op	Inherent ~	Inherent #	Description	5 H	3 N	2 Z	1 V	0 C
LSL	LSLA													48	2	1	A } c ← b7 … b0 ← 0	•	↕	↕	↕	↕
	LSLB													58	2	1	B }	•	↕	↕	↕	↕
	LSL				08	6	2	68	6+	2+	78	7	3				M }	•	↕	↕	↕	↕
LSR	LSRA													44	2	1	A } 0 → b7 … b0 → c	•	0	↕	•	↕
	LSRB													54	2	1	B }	•	0	↕	•	↕
	LSR				04	6	2	64	6+	2+	74	7	3				M }	•	0	↕	•	↕
MUL														3D	11	1	A × B − D (Unsigned)	•	•	↕	•	9
NEG	NEGA													40	2	1	$\overline{A}$ + 1 − A	8	↕	↕	↕	↕
	NEGB													50	2	1	$\overline{B}$ + 1 − B	8	↕	↕	↕	↕
	NEG				00	6	2	60	6+	2+	70	7	3				$\overline{M}$ + 1 − M	8	↕	↕	↕	↕
NOP														12	2	1	No Operation	•	•	•	•	•
OR	ORA	BA	2	2	9A	4	2	AA	4+	2+	BA	5	3				A V M − A	•	↕	↕	0	•
	ORB	CA	2	2	DA	4	2	EA	4+	2+	FA	5	3				B V M − B	•	↕	↕	0	•
	ORCC	1A	3	2													CC V IMM − CC				7	
PSH	PSHS	34	5+[4]	2													Push Registers on S Stack	•	•	•	•	•
	PSHU	36	5+[4]	2													Push Registers on U Stack	•	•	•	•	•
PUL	PULS	35	5+[4]	2													Pull Registers from S Stack	•	•	•	•	•
	PULU	37	5+[4]	2													Pull Registers from U Stack	•	•	•	•	•
ROL	ROLA													49	2	1	A } c ← b7 … b0 ← (c)	•	↕	↕	↕	↕
	ROLB													59	2	1	B }	•	↕	↕	↕	↕
	ROL				09	6	2	69	6+	2+	79	7	3				M }	•	↕	↕	↕	↕
ROR	RORA													46	2	1	A } c → b7 … b0 → (c)	•	↕	↕	•	↕
	RORB													56	2	1	B }	•	↕	↕	•	↕
	ROR				06	6	2	66	6+	2+	76	7	3				M }	•	↕	↕	•	↕
RTI														3B	6 15	1	Return From Interrupt					7
RTS														39	5	1	Return from Subroutine	•	•	•	•	•
SBC	SBCA	B2	2	2	92	4	2	A2	4+	2+	B2	5	3				A M C − A	8	↕	↕	↕	↕
	SBCB	C2	2	2	D2	4	2	E2	4+	2+	F2	5	3				B M - C − B	8	↕	↕	↕	↕
SEX														1D	2	1	Sign Extend B into A	•	↕	↕	0	•
ST	STA				97	4	2	A7	4+	2+	B7	5	3				A − M	•	↕	↕	0	•
	STB				D7	4	2	E7	4+	2+	F7	5	3				B − M	•	↕	↕	0	•
	STD				DD	5	2	ED	5+	2+	FD	6	3				D − M M + 1	•	↕	↕	0	•
	STS				10 DF	6	3	10 EF	6+	3+	10 FF	7	4				S − M M + 1	•	↕	↕	0	•
	STU				DF	5	2	EF	5+	2+	FF	6	3				U − M M + 1	•	↕	↕	0	•
	STX				9F	5	2	AF	5+	2+	BF	6	3				X − M M + 1	•	↕	↕	0	•
	STY				10 9F	6	3	10 AF	6+	3+	10 BF	7	4				Y − M M + 1	•	↕	↕	0	•
SUB	SUBA	80	2	2	90	4	2	A0	4+	2+	B0	5	3				A M − A	8	↕	↕	↕	↕
	SUBB	C0	2	2	D0	4	2	E0	4+	2+	F0	5	3				B - M − B	8	↕	↕	↕	↕
	SUBD	83	4	3	93	6	2	A3	6+	2+	B3	7	3				D M M + 1 − D	•	↕	↕	↕	↕
SWI	SWI[6]													3F	19	1	Software Interrupt 1	•	•	•	•	•
	SWI[6]													10 3F	20	2	Software Interrupt 2	•	•	•	•	•
	SWI[6]													11 3F	20	1	Software Interrupt 3	•	•	•	•	•
SYNC														13	≥4	1	Synchronize to Interrupt	•	•	•	•	•
TFR	R1 R2													1F	6	2	R1 − R2[2]	•	•	•	•	•
TST	TSTA													4D	2	1	Test A	•	↕	↕	0	•
	TSTB													5D	2	1	Test B	•	↕	↕	0	•
	TST				0D	6	2	6D	6+	2+	7D	7	3				Test M	•	↕	↕	0	•

Notes

1. This column gives a base cycle and byte count. To obtain total count, add the values obtained from the INDEXED ADDRESSING MODE table, Table 2
2. R1 and R2 may be any pair of 8 bit or any pair of 16 bit registers
   The 8 bit registers are A, B, CC, DP
   The 16 bit registers are X, Y, U, S, D, PC
3. EA is the effective address
4. The PSH and PUL instructions require 5 cycles plus 1 cycle for each byte pushed or pulled
5. 5(6) means 5 cycles if branch not taken, 6 cycles if taken (Branch instructions)
6. SWI sets I and F bits. SWI2 and SWI3 do not affect I and F
7. Conditions Codes set as a direct result of the instruction
8. Value of half-carry flag is undefined.
9. Special Case − Carry set if b7 is SET

Courtesy of Motorola, Inc.

APPENDICE B
INSTRUCTIONS DE BRANCHEMENT DU 6809

Instruction	Forms	Addressing Mode Relative OP	- 5	#	Description	5 H	3 N	2 Z	1 V	0 C
BCC	BCC	24	3	2	Branch C = 0	•	•	•	•	•
	LBCC	10 24	5(6)	4	Long Branch C = 0	•	•	•	•	•
BCS	BCS	25	3	2	Branch C = 1	•	•	•	•	•
	LBCS	10 25	5(6)	4	Long Branch C = 1	•	•	•	•	•
BEQ	BEQ	27	3	2	Branch Z = 0	•	•	•	•	•
	LBEQ	10 27	5(6)	4	Long Branch Z = 0	•	•	•	•	•
BGE	BGE	2C	3	2	Branch ≥ Zero	•	•	•	•	•
	LBGE	10 2C	5(6)	4	Long Branch ≥ Zero	•	•	•	•	•
BGT	BGT	2E	3	2	Branch > Zero	•	•	•	•	•
	LBGT	10 2E	5(6)	4	Long Branch > Zero	•	•	•	•	•
BHI	BHI	22	3	2	Branch Higher	•	•	•	•	•
	LBHI	10 22	5(6)	4	Long Branch Higher	•	•	•	•	•
BHS	BHS	24	3	2	Branch Higher or Same	•	•	•	•	•
	LBHS	10 24	5(6)	4	Long Branch Higher or Same	•	•	•	•	•
BLE	BLE	2F	3	2	Branch ≤ Zero	•	•	•	•	•
	LBLE	10 2F	5(6)	4	Long Branch ≤ Zero	•	•	•	•	•
BLO	BLO	25	3	2	Branch lower	•	•	•	•	•
	LBLO	10 25	5(6)	4	Long Branch Lower	•	•	•	•	•

Instruction	Forms	Addressing Mode Relative OP	- 5	#	Description	5 H	3 N	2 Z	1 V	0 C
BLS	BLS	23	3	2	Branch Lower or Same	•	•	•	•	•
	LBLS	10 23	5(6)	4	Long Branch Lower or Same	•	•	•	•	•
BLT	BLT	2D	3	2	Branch < Zero	•	•	•	•	•
	LBLT	10 2D	5(6)	4	Long Branch < Zero	•	•	•	•	•
BMI	BMI	2B	3	2	Branch Minus	•	•	•	•	•
	LBMI	10 2B	5(6)	4	Long Branch Minus	•	•	•	•	•
BNE	BNE	26	3	2	Branch Z ≠ 0	•	•	•	•	•
	LBNE	10 26	5(6)	4	Long Branch Z ≠ 0	•	•	•	•	•
BPL	BPL	2A	3	2	Branch Plus	•	•	•	•	•
	LBPL	10 2A	5(6)	4	Long Branch Plus	•	•	•	•	•
BRA	BRA	20	3	2	Branch Always	•	•	•	•	•
	LBRA	16	5	3	Long Branch Always	•	•	•	•	•
BRN	BRN	21	3	2	Branch Never	•	•	•	•	•
	LBRN	10 21	5	4	Long Branch Never	•	•	•	•	•
BSR	BSR	8D	7	2	Branch to Subroutine	•	•	•	•	•
	LBSR	17	9	3	Long Branch to Subroutine	•	•	•	•	•
BVC	BVC	28	3	2	Branch V = 0	•	•	•	•	•
	LBVC	10 28	5(6)	4	Long Branch V = 0	•	•	•	•	•
BVS	BVS	29	3	2	Branch V = 1	•	•	•	•	•
	LBVS	10 29	5(6)	4	Long Branch V = 1	•	•	•	•	•

SIMPLE BRANCHES

	OP	~	#
BRA	20	3	2
LBRA	16	5	3
BRN	21	3	2
LBRN	1021	5	4
BSR	8D	7	2
LBSR	17	9	3

SIMPLE CONDITIONAL BRANCHES (Notes 1-4)

Test	True	OP	False	OP
N = 1	BMI	2B	BPL	2A
Z = 1	BEQ	27	BNE	26
V = 1	BVS	29	BVC	28
C = 1	BCS	25	BCC	24

SIGNED CONDITIONAL BRANCHES (Notes 1-4)

Test	True	OP	False	OP
r > m	BGT	2E	BLE	2F
r ≥ m	BGE	2C	BLT	2D
r = m	BEQ	27	BNE	26
r ≤ m	BLE	2F	BGT	2E
r < m	BLT	2D	BGE	2C

UNSIGNED CONDITIONAL BRANCHES (Notes 1-4)

Test	True	OP	False	OP
r > m	BHI	22	BLS	23
r ≥ m	BHS	24	BLO	25
r = m	BEQ	27	BNE	26
r ≤ m	BLS	23	BHI	22
r < m	BLO	25	BHS	24

Notes

1. All conditional branches have both short and long variations
2. All short branches are 2 bytes and require 3 cycles.
3. All conditional long branches are formed by prefixing the short branch opcode with $10 and using a 16-bit destination offset
4. All conditional long branches require 4 bytes and 6 cycles if the branch is taken or 5 cycles if the branch is not taken
5. 5(6) means. 5 cycles if branch not taken, 6 cycles if taken

APPENDICE C
POST-OCTETS EN MODE D'ADRESSAGE INDEXÉ

Type	Forms	Non Indirect				Indirect			
		Assembler Form	Postbyte OP Code	+ ~	+ #	Assembler Form	Postbyte OP Code	+ ~	+ #
Constant Offset From R (2's Complement Offsets)	No Offset	,R	1RR00100	0	0	[,R]	1RR10100	3	0
	5 Bit Offset	n, R	0RRnnnnn	1	0	defaults to 8-bit			
	8 Bit Offset	n, R	1RR01000	1	1	[n, R]	1RR11000	4	1
	16 Bit Offset	n, R	1RR01001	4	2	[n, R]	1RR11001	7	2
Accumulator Offset From R (2's Complement Offsets)	A Register Offset	A, R	1RR00110	1	0	[A, R]	1RR10110	4	0
	B Register Offset	B, R	1RR00101	1	0	[B, R]	1RR10101	4	0
	D Register Offset	D, R	1RR01011	4	0	[D, R]	1RR11011	7	0
Auto Increment/ Decrement R	Increment By 1	,R+	1RR00000	2	0	not allowed			
	Increment By 2	,R++	1RR00001	3	0	[,R++]	1RR10001	6	0
	Decrement By 1	,−R	1RR00010	2	0	not allowed			
	Decrement By 2	,−−R	1RR00011	3	0	[,−−R]	1RR10011	6	0
Constant Offset From PC (2's Complement Offsets)	8 Bit Offset	n, PCR	1xx01100	1	1	[n, PCR]	1xx11100	4	1
	16 Bit Offset	n, PCR	1xx01101	5	2	[n, PCR]	1xx11101	8	2
Extended Indirect	16 Bit Address	−	−	−	−	[n]	10011111	5	2

R = X, Y, U or S
x = Don't Care

RR:
00=X
01=Y
10=U
11=S

$\underset{\sim}{+}$ and + indicate the number of additional cycles and bytes for the particular variation.

Courtesy of Motorola, Inc.

APPENDICE D
TABLE DE CONVERSION HEXADÉCIMALE

HEX	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	00	000
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	0	0
1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	256	4096
2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	512	8192
3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	768	12288
4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	1024	16384
5	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	1280	20480
6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111	1536	24576
7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127	1792	28672
8	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143	2048	32768
9	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159	2304	36864
A	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175	2560	40960
B	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191	2816	45056
C	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207	3072	49152
D	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223	3328	53248
E	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239	3584	57344
F	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255	3840	61440

5		4		3		2		1		0	
HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC
0	0	0	0	0	0	0	0	0	0	0	0
1	1,048,576	1	65,536	1	4,096	1	256	1	16	1	1
2	2,097,152	2	131,072	2	8,192	2	512	2	32	2	2
3	3,145,728	3	196,608	3	12,288	3	768	3	48	3	3
4	4,194,304	4	262,144	4	16,384	4	1,024	4	64	4	4
5	5,242,880	5	327,680	5	20,480	5	1,280	5	80	5	5
6	6,291,456	6	393,216	6	24,576	6	1,536	6	96	6	6
7	7,340,032	7	458,752	7	28,672	7	1,792	7	112	7	7
8	8,388,608	8	524,288	8	32,768	8	2,048	8	128	8	8
9	9,437,184	9	589,824	9	36,864	9	2,304	9	144	9	9
A	10,485,760	A	655,360	A	40,960	A	2,560	A	160	A	10
B	11,534,336	B	720,896	B	45,056	B	2,816	B	176	B	11
C	12,582,912	C	786,432	C	49,152	C	3,072	C	192	C	12
D	13,631,488	D	851,968	D	53,248	D	3,328	D	208	D	13
E	14,680,064	E	917,504	E	57,344	E	3,584	E	224	E	14
F	15,728,640	F	983,040	F	61,440	F	3,840	F	240	F	15

TABLE DES MATIÈRES

LA BIBLIOTHÈQUE SYBEX

OUVRAGES GÉNÉRAUX

VOTRE PREMIER ORDINATEUR *par* *RODNAY ZAKS,*
296 pages, Réf. 394

VOTRE ORDINATEUR ET VOUS *par* *RODNAY ZAKS,*
296 pages, Réf. 271

DU COMPOSANT AU SYSTÈME : une introduction aux microprocesseurs *par* *RODNAY ZAKS,*
636 pages, Réf. 340

TECHNIQUES D'INTERFACE aux microprocesseurs *par* *AUSTIN LESEA ET RODNAY ZAKS,*
450 pages, Réf. 339, 3ème édition

LEXIQUE INTERNATIONAL MICROORDINATEURS, avec dictionnaire abrégé en 10 langues
192 pages, Réf. 234

GUIDE DES MICRO-ORDINATEURS A MOINS 3 000 F *par* *JOËL PONCET,*
144 pages, Réf. 322

LEXIQUE MICRO-INFORMATIQUE *par* *PIERRE LE BEUX,*
140 pages, Réf. 369

BASIC

VOTRE PREMIER PROGRAMME BASIC *par* *RODNAY ZAKS,*
208 pages, Réf. 263

INTRODUCTION AU BASIC *par* *PIERRE LE BEUX,*
336 pages, Réf. 335

LE BASIC PAR LA PRATIQUE : 60 exercices *par* *JEAN-PIERRE LAMOITIER,*
252 pages, Réf. 395

LE BASIC POUR L'ENTREPRISE *par* *XUAN TUNG BUI,*
204 pages, Réf. 253, 2ème édition

PROGRAMMES EN BASIC, Mathématiques, Statistiques, Informatique *par* *ALAN R. MILLER,*
318 pages, Réf. 259

AU COEUR DES JEUX EN BASIC *par* *RICHARD MATEOSIAN,*
352 pages, Réf. 233

JEUX D'ORDINATEUR EN BASIC *par* *DAVID H. AHL,*
192 pages, Réf. 246

NOUVEAUX JEUX D'ORDINATEUR EN BASIC *par* *DAVID H. AHL,*
204 pages, Réf. 247

PASCAL

INTRODUCTION AU PASCAL *par* *PIERRE LE BEUX,*
496 pages, Réf. 330

LE PASCAL PAR LA PRATIQUE *par* *PIERRE LE BEUX ET HENRI TAVERNIER,*
562 pages, Réf. 361

LE GUIDE DU PASCAL *par* *JACQUES TIBERGHIEN,*
504 pages, Réf. 232

PROGRAMMES EN PASCAL pour Scientifiques et Ingénieurs *par* *ALAN R. MILLER,*
392 pages, Réf. 240

AUTRES LANGAGES

INTRODUCTION A ADA *par Pierre Le Beux,*
366 pages, Réf. 360

MICROORDINATEUR

ALICE

JEUX EN BASIC POUR ALICE *par Pierre Monsaut,*
96 pages, Réf. 320

APPLE / MACINTOSH

PROGRAMMEZ EN BASIC SUR APPLE II *par Léopold Laurent,*
208 pages, Réf. 333

APPLE II 66 PROGRAMMES BASIC *par Stanley R. Trost,*
192 pages, Réf. 283

JEUX EN PASCAL SUR APPLE *par Douglas Hergert et Joseph T. Kalash,*
372 pages, Réf. 241

GUIDE DU BASIC APPLE II *par Douglas Hergert,*
272 pages, Réf. 306

APPLE II, PREMIERS PROGRAMMES *par Rodnay Zaks,*
248 pages, Réf. 373

MACINTOSH, GUIDE DE L'UTILISATEUR *par Joseph Caggiano,*
208 pages, Réf. 396

ATARI

JEUX EN BASIC SUR ATARI *par Paul bunn,*
96 pages, Réf. 282

ATMOS

JEUX EN BASIC SUR ATMOS *par Pierre Monsaut,*
96 pages, Réf. 346

ATMOS, 56 PROGRAMMES *par Stanley R. Trost,*
180 pages, Réf. 372

COMMODORE 64

JEUX EN BASIC SUR COMMODORE 64 *par Pierre Monsaut,*
96 pages, Réf. 317

COMMODORE 64, PREMIERS PROGRAMMES *par Rodnay Zaks,*
248 pages, Réf. 342

GUIDE DU BASIC VIC 20, COMMODORE 64 *par Douglas Hergert,*
240 pages, Réf. 312

COMMODORE 64, GUIDE DE L'UTILISATEUR *par J. Kassmer,*
144 pages, Réf. 314

COMMODORE 64, 66 PROGRAMMES *par Stanley R. Trost,*
192 pages, Réf. 319

DRAGON

JEUX EN BASIC SUR DRAGON *par Pierre Monsaut,*
96 pages, Réf. 324

GOUPIL

PROGRAMMEZ VOS JEUX SUR GOUPIL *par François Abella,*
208 pages, Réf. 264

IBM

IBM PC EXERCICES EN BASIC *par Jean-Pierre Lamoitier,*
256 pages, Réf. 338

IBM PC Guide de l'utilisateur *par Joan Lasselle et Carol Ramsey,*
160 pages, Réf. 301

IBM PC 66 PROGRAMMES BASIC *par Stanley R. Trost,*
192 pages, Réf. 280

LASER

LASER JEUX D'ACTION *par Pierre Monsaut,*
96 pages, Réf. 371

MO5

MO5 JEUX D'ACTION *par Pierre Monsaut,*
96 pages, Réf. 367

MO5, PREMIERS PROGRAMMES *par Rodnay Zaks,*
248 pages, Réf. 370

ORIC

JEUX EN BASIC SUR ORIC *par Peter Shaw,*
96 pages, Réf. 278

ORIC PREMIERS PROGRAMMES *par Rodnay Zaks,*
248 pages, Réf. 344

SHARP

DÉCOUVREZ LE SHARP PC-1500 ET LE TRS-80 PC-2 *par Michel Lhoir,*
2 tomes, Réf. 261-262

SPECTRUM

PROGRAMMEZ EN BASIC SUR SPECTRUM *par S.M. Gee,*
208 pages, Réf. 252

JEUX EN BASIC SUR SPECTRUM *par Peter Shaw,*
96 pages, Réf. 276

SPECTRUM, PREMIERS PROGRAMMES *par Rodnay Zaks,*
248 pages, Réf. 381

SPECTRUM JEUX D'ACTION *par Pierre Monsaut,*
96 pages, Réf. 368

TI 99/4

PROGRAMMEZ VOS JEUX SUR TI 99/4 *par François Abella,*
160 pages, Réf. 303

TO 7

JEUX EN BASIC SUR TO 7 *par Pierre Monsaut,*
96 pages, Réf. 326

TO 7, PREMIERS PROGRAMMES *par Rodnay Zaks,*
248 pages, Réf. 328

TIMEX

ZX 81 GUIDE DE L'UTILISATEUR *par Douglas Hergert,*
208 pages, Réf. 256

TRS-80

PROGRAMMEZ EN BASIC SUR TRS-80 *par Léopold Laurent,*
2 tomes, Réf. 250-251

DÉCOUVREZ LE SHARP PC-1500 ET LE TRS-80 PC-2 *par Michel Lhoir,*
2 tomes, Réf. 366-262

JEUX EN BASIC SUR TRS-80 MC-10 *par Pierre Monsaut,*
96 pages, Réf. 323

JEUX EN BASIC SUR TRS-80 *par Chris Palmer,*
96 pages, Réf. 302

JEUX EN BASIC SUR TRS-80 COULEUR *par Pierre Monsaut,*
96 pages, Réf. 325

TRS-80 MODÈLE 100, GUIDE DE L'UTILISATEUR *par Orson Kellog,*
112 pages, Réf. 300

VIC 20

PROGRAMMEZ EN BASIC SUR VIC 20 *par G. O. Hamann,*
2 tomes, Réf. 244-337

JEUX EN BASIC SUR VIC 20 *par Alastair Gourlay,*
96 pages, Réf. 277

VIC 20, PREMIERS PROGRAMMES *par Rodnay Zaks,*
248 pages, Réf. 341

GUIDE DU BASIC VIC 20, COMMODORE 64 *par Douglas Hergert,*
240 pages, Réf. 312

VIC 20 JEUX D'ACTION *par Pierre Monsaut,*
96 pages, Réf. 345

ZX 81

ZX 81 GUIDE DE L'UTILISATEUR *par Douglas Hergert,*
208 pages, Réf. 256

ZX 81 56 PROGRAMMES BASIC *par Stanley R. Trost,*
192 pages, Réf. 304

GUIDE DU BASIC ZX 81 *par Douglas Hergert,*
204 pages, Réf. 285

JEUX EN BASIC SUR ZX 81 *par Mark Charlton,*
96 pages, Réf. 275

ZX 81 PREMIERS PROGRAMMES *par Rodnay Zaks,*
248 pages, Réf. 343

MICRO-PROCESSEURS

PROGRAMMATION DU Z80 *par Rodnay Zaks,*
618 pages, Réf. 220

APPLICATIONS DU Z80 *par James W. Coffron,*
304 pages, Réf. 274

PROGRAMMATION DU 6502 *par Rodnay Zaks,*
376 pages, Réf. 331, 2ème édition

APPLICATIONS DU 6502 *par Rodnay Zaks,*
288 pages, Réf. 332

PROGRAMMATION DU 6800 *par Daniel-Jean David et Rodnay Zaks,*
374 pages, Réf. 327

PROGRAMMATION DU 6809 *par Rodnay Zaks et William Labiak,*
392 pages, Réf. 328

PROGRAMMATION DU 8086/8088 *par James W. Coffron,*
304 pages, Réf. 316

MISE EN OEUVRE DU 68000 *par C. Vieillefond,*
352 pages, Réf. 363

SYSTÈMES D'EXPLOITATION

GUIDE DU CP/M AVEC MP/M *par Rodnay Zaks,*
354 pages, Réf. 336

CP/M APPROFONDI *par Alan R. Miller,*
380 pages, Réf. 334

INTRODUCTION AU p-SYSTEM UCSD *par Charles W. Grant et Jon Butah,*
308 pages, Réf. 365

GUIDE DU PC DOS *par Richard A. King,*
240 pages, Réf. 313

LOGICIELS ET APPLICATIONS

INTRODUCTION AU TRAITEMENT DE TEXTE *par Hal Glatzer,*
228 pages, Réf. 243

INTRODUCTION A WORDSTAR *par Arthur Naiman,*
200 pages, Réf. 255

WORDSTAR APPLICATIONS *par Julie Anne Arca,*
320 pages, Réf. 305

VISICALC APPLICATIONS *par Stanley R. Trost,*
304 pages, Réf. 258

VISICALC POUR L'ENTREPRISE *par Dominique Helle,*
304 pages, Réf. 309

INTRODUCTION A dBASE II *par Alan Simpson,*
280 pages, Réf. 364

DE VISICALC A VISI ON *par Jacques Bourdeu,*
256 pages, Réf. 321

La plupart de ces ouvrages existent en version anglaise. N'hésitez pas à demander notre catalogue.

EN ANGLAIS

BASIC EXERCISES FOR APPLE *by* *Jean-Pierre Lamoitier,*
232 pages, Réf. 0-084

BASIC FOR BUSINESS *by* *Douglas Hergert,*
224 pages, Réf. 0-080

CELESTIAL BASIC : Astronomy on your Computer *by* *Eric Burgess,*
228 pages, Réf. 0-087

INTRODUCTION TO PASCAL (Including UCSD Pascal) *by* *Rodnay Zaks,*
422 pages, Réf. 0-066

DOING BUSINESS WITH PASCAL *by* *Richard Hergert and Douglas Hergert,*
380 pages, Réf. 0-091

MASTERING VISICALC *by* *Douglas Hergert,*
224 pages, Réf. 0-090

THE APPLE CONNECTION *by* *James W. Coffron,*
228 pages, Réf. 0-085

PROGRAMMING THE Z8000 *by* *Richard Mateosian,*
300 pages, Réf. 0-032

A MICROPROGRAMMED APL IMPLEMENTATION *by* *Rodnay Zaks,*
350 pages, Réf. 0-005

ADVANCED 6502 PROGRAMMING *by* *Rodnay Zaks,*
292 pages, Réf. 0-089

FORTRAN PROGRAMS FOR SCIENTISTS AND ENGINEERS *by* *Alan R. Miller,*
320 pages, Réf. 0-082

POUR UN CATALOGUE COMPLET DE NOS PUBLICATIONS

FRANCE
6-8, Impasse du Curé
75881 PARIS CEDEX 18
Tél. : (1) 203.95.95
Télex : 211801

U.S.A.
2344 Sixth Street
Berkeley, CA 94710
Tel. : (415) 848.8233
Telex : 336311

ALLEMAGNE
Volgelsanger. WEG 111
4000 Düsseldorf 30
Post Bos N° 30.09.61
Tel. : (0211) 626441
Telex : 08588163

ANGLETERRE
Unit 4 - Bourne Industrial Park
Bourne Road, Crayford
Kent DA1 4BU
Tel. : (0322) 57717



Paris • Berkeley • Düsseldorf • Londres